Efficient Elimination of False Positives Using Bounded Model Checking

Tukaram Muske, Advaita Datar, Mayur Khanzode, Kumar Madhukar
TRDDC, Tata Consultancy Services,
Pune, India
{t.muske, advaita.datar, mayur.khanzode, kumar.madhukar}@tcs.com

***Abstract*—Software verification using abstract interpretation is scalable but imprecise. Model checking is precise in verifying a property but not scalable. Often, these two techniques are combined to achieve better precision. A possible way is to analyze a software system first by using abstract interpretation and later eliminating the false positives using bounded model checking. This is a time consuming process as it typically involves verifying an assertion corresponding to each generated warning. We observe verifying all assertions often introduces redundancy, and some verifications may not even eliminate a false positive. In this paper, we present an approach consisting of three techniques to make such false positives elimination faster. Two of the techniques identify an assertion as being equivalent to an other assertion thus avoiding its verification. The third technique tries to identify and skip a class of assertion verifications that will not eliminate a false positive. Empirical results indicate that these techniques are quite useful in reducing the number of assertions being verified by 53%, and the false positives elimination time by 60%.**

***Keywords*—Abstract Interpretation; Model Checking; False Positives Elimination; Data Flow Analysis.**

I. INTRODUCTION

Software verification using abstract interpretation [1] has been effective in proving the absence of runtime errors such as Division by Zero (ZD), Array Index Out of Bound (AIOB) and buffer overflow. It has successfully been used to verify very large software systems, but generates too many false warnings, commonly referred to as false positives [2]. On the other hand, model checking is precise for property verification, but it often faces the state explosion problem as programs include unbounded-loops, recursions, complex data structures [3][4]. Under these circumstances, a property verification may not succeed and it is a concern.

There have been several attempts at combining these two techniques to improve precision [5][6][7], i.e., to generate fewer warnings which, in turn, would reduce the cost of their manual review. Abstract interpretation, being light weight, is used first and then generated warnings are processed by a model checker to eliminate false positives [8][9]. This processing includes generation of an assertion corresponding to each warning and its verification using a model checker. If an assertion is successfully verified, its corresponding warning is a false positive and is eliminated.

This process of False Positives Elimination (FPE) is very time consuming as each assertion needs to be verified by a model checker, and an average assertion verification time is significant [9][10]. Hence, there is a need to make it faster. To the best of our knowledge, nothing has been done to make such false positives elimination efficient. In practice, we observe verifying all assertions often introduce redundancies, and few verifications even may not eliminate a false positive. We use these observations to make FPE efficient by reducing the number of assertions being verified.

Throughout this paper:

1) we use A_n to denote the assertion at line *n* and $V(A_n)$ to denote its verification.
2) in our examples, for clarity of representation, we have eliminated some parts of the code (like assigning non-deterministic values to input variables, conditional pre-processor code to include single assertion at a time).
3) we describe FPE more particularly, using CBMC (C Bounded Model Checker) [11]. We chose CBMC for our experimentation because we have prior experience in its usage and it is integrated in the existing tool set [9].

Consider the motivating example in Figure 1, where each access of an array is reported as a warning by abstract interpretation (AIOB property), since their index values are unknown for abstract interpretation. This example also shows six assertions, one corresponding to each of the warnings. Verifications of all these six assertions is not required since some of them are redundant or non-verifiable. This is explained as below:

1) A_{15} and A_{17} are equivalent since their assert expressions are same and *n* is not modified in between. With this equivalence, $V(A_{17})$ is found redundant as it has same result as that of $V(A_{15})$. That is, if A_{15} is verified successfully, warnings related to A_{15} and A_{17} will be eliminated, else they will continue to be warnings after FPE.
2) If *Overflow-Underflow* (OFUF) property is proved on the code, $V(A_{21})$ has same result as that of $V(A_{23})$, because the involved *index* variable has an *unsigned* data type. That is, success in $V(A_{23})$ guarantees success in $V(A_{21})$, whereas if $V(A_{23})$ fails then $V(A_{21})$ is most likely to fail. Thus, their corresponding warnings will together be either false positives or continue to be warnings after FPE. This observation finds the $V(A_{21})$ is redundant.
3) $V(A_{35})$ and $V(A_{39})$ require analysis of the unbounded loop (run time dependent loop at line 34), and hence, these verifications either will terminate with *out of memory* or will produce an error trace depicting insufficient loop unwinding. Please refer to Section II-A for more details on insufficient loop unwinding by CBMC. These assertions

 ISBN: 978-1-61208-307-0

```
    int rColors[10], gColors[10];

11.   void f1(int r, int g){
12.      unsigned int i = 0, n, temp;
13.
14.      n = ...;
15.      assert( n>=0 && n<10 );
16.      rColors[n] = r;
17.      assert( n>=0 && n<10 );
18.      gColors[n] = g;
19.
20.     i = ...;
21.     assert(temp=i++, temp>=0 && temp<10 );
22.     rColors[i++] = ...;
23.     assert(temp=i++, temp>=0 && temp<10 );
24.     gColors[i++] = ...;
      }
```

```
    char *str, charArr[20];

31.   void f2(){
32.     int i = 0;
33.     ...
34.     while( *str != ' ' ){
35.         assert( i>=0 && i<20 );
36.         charArr[i] = *str;
37.         i++;
38.     }
39.     assert( i>=0 && i<20 );
40.     charArr[i] = '\0';
        ...
      }
```

Figure 1: Example and Annotation - 1

can not be verified, so are non-verifiable; we are unable to eliminate a false positive.

We propose three techniques for efficient FPE.

1) Property Independent Redundant Assertion Identification Technique (PI-RAIT): It partitions assertions and selects a leader assertion for each partition such that if leader assertion hold so does the other assertions from its partition. A partition so formed represents a set of either false positives or the warnings together, and it requires only the leader assertion to be verified. This technique will put A_{15} and A_{17} in the same partition with A_{15} as the leader.
2) Property Dependent Redundant Assertion Identification Technique (PD-RAIT): It includes partitioning of assertions similar in PI-RAIT, but depends on the characteristic of a run-time property. It only differs with PI-RAIT in the way of identifying equivalence of assertions, where it uses property characteristics and practical observations (code patterns) to identify the equivalence of assertions. This technique will put A_{21} and A_{23} in the same partition with A_{23} as the leader.
3) Non-Verifiable Assertion Identification Technique (NVAIT): It includes identifying assertion verifications which require analysis of unbounded loops and most likely they can not be verified successfully. This technique will identify A_{35} and A_{39} as *Non-Verifiable Assertions* (NVAs).

Out of six assertions in Figure 1, these proposed techniques identify only two assertions (A_{15} and A_{23}) to be verified and skip the other four. This makes FPE considerably faster. We applied this approach in different FPE settings for two automotive industry C applications and the results indicate that these techniques reduce the number of assertions being verified by 53% and the resultant FPE time by 60%.

We discuss in detail PI-RAIT in Section II, and PD-RAIT and NVAIT in Section III. The implementation and empirical results are described in Section IV. Section V presents related work, and finally, we conclude with future work in Section VI.

II. Property Independent RAIT

This section discusses in detail the proposed Property Independent RAIT (PI-RAIT) and presents an algorithm for it.

A. CBMC

In this paper, we describe FPE and the proposed techniques using CBMC, hence it is briefly described. CBMC is a Bounded Model Checker for ANSI-C and C++ programs. It takes an entry function and a property to be verified that is expressed as an assertion. The specified entry function represents a context at which the assertion is verified and can be an entry point of the application or any other function having the input assertion. If an assertion holds for all execution paths, CBMC reports verification success. If it does not hold, generates an error trace leading to the property violation. The verification is performed by unwinding the loops in the program, and it is necessary for all loops to have a finite upper bound [4]. For unbounded loops, it takes a user provided bound (unwinding count) as an upper bound. The provided bound should be enough to capture the program semantics, so that the property verification is sound and complete. Further, when the bound is not enough, it produces a trace (*loop unwinding counterexample*) to demonstrate the insufficiency of loop unwinding.

B. Reduction of Assertions in FPE: Need

Abstract interpretation usually reports a large number of analysis warnings [4]. In a FPE process, ideally, an assertion corresponding to each generated warning should be verified with application entry point as the entry function. However, it often does not scale or generates a loop unwinding counterexample. To overcome this problem, a different approach of incremental code context (*context expansion*) is adopted [8]. In this approach, an assertion verification is started with a minimal code context only, i.e., the enclosed function of the assertion. Later, the context is incremented to the callers of the enclosed function until one of the following holds:

1) its corresponding false positive is eliminated
2) context reaches the application entry
3) a certain time limit is achieved

 ISBN: 978-1-61208-307-0

FPE with code context expansion, as compared to FPEs at function and application levels, eliminates more false positives but involves verifying an assertion multiple times. Performing numerous verifications for each of the generated assertion increases FPE time even further. Hence, there is a need to minimize the number of assertions being verified.

C. Equivalence of Assertions

We have observed that, in practice, many of the generated warnings are similar, and hence, their corresponding assertions are also likely to be equivalent. The code snippet in Figure 2 depicts three Zero Division (ZD) warnings and their corresponding assertions. These warnings are similar, because the denominator in each ZD warning is the same variable taking values from the same source. They together represent a class of false positives or an error. Hence, the added assertions are also equivalent.

```
11.      denom = ...;
12.      if(...){
13.          assert(denom != 0); r1 = n1/denom;
14.      }
15.
16.     assert(denom != 0); r2 = n2/denom;
17.
18.      if(...){
19.          assert(denom != 0); r3 = n3/denom;
20.      }
```

Figure 2: Example and Annotation - 2

More precisely, two assertions are equivalent if -

1) their assert expressions are structurally similar *and*
2) the variables referred to by these assertions have the same source for their values.

The structural similarity of expressions requires that variables used, the operators and their order of appearance in the expression be the same. For example, given two ZD assertions with their expressions as

- $(a+b+c)! = 0$ and $(a+b+c)! = 0$ are potentially equivalent.
- $(v+1)! = 0$ and $(1+v)! = 0$ are not equivalent, since operands appear in different order.
- $(v1 + func())! = 0$ and $(v1 + func())! = 0$ are not equivalent since different calls to a function can return different values.

D. Partitioning of Assertions

We use the equivalence of assertions to reduce the number of FPE assertion verifications. The equivalent assertions are put in the same partition. An assertion in a partition is tagged as a *Leader Assertion* (LA) only if its successful verification by model checker guarantees the successful verification of other assertions in its partition. This ensures that the warnings corresponding to assertions in a partition are regarded as false positives when the leader of the partition is verified successfully. In the other case, if the leader verification fails then verification of other assertions in that partition is most likely to fail. Essentially, the verification result of assertions in a partition follow the verification result of the partition leader, hence, they are referred to as *Follower Assertions* (FAs). If an assertion can not be equivalent to an other assertion, it will be the only member (LA) of its partition without having the follower(s). Thus, in a partition there will be strictly only one LA and any number of FAs including zero.

Using this approach, burden of eliminating a false positive corresponding to a FA is transferred to its leader, and hence, there is a chance that it might miss on eliminating a false positive. This is because, there could be a scenario in which a FA is able to identify a false positive but its leader is not. We permit this assuming such scenarios would be very rare in practice.

We tag an assertion A in a partition as a LA, only if all paths reaching any other assertion (FA) also go through A. Under this criterion, A_{16} is selected as a LA for the partition of the equivalent assertions in Figure 2. Other assertion (A_{13} or A_{19}), can not be tagged as a LA since there exists a path reaching A_{16} but not going through it.

E. Assertions Partitioning Algorithm

We use *must reachability* and *must liveness* of assertions to compute the LAs and associate them with their corresponding FAs. The reaching and live assertions being computed denote *must* data flow information, i.e., they represent the assertions that are definitely reaching or definitely live from the program points at which they appear. These *must reaching (must live)* assertions are similar to *reaching definitions (live variables)* [12], with assertions replacing variables and *must* information replacing *may*. It should be noted that the assertions under consideration are unique in themselves where they are uniquely identified by the program points at which they appear.

1) Must Reaching Assertion (MRA): An assertion with expression e at a program point P is a MRA at its succeeding program point P' if every path coming to P' passes through P and, no path segment between P and P' contains a *l-value* occurrence of any of the *r-value(s)* of e. This ensures that each execution path through P' also includes P.

With a slight abuse of notation, we denote a program point of an assertion by the assertion itself. In Figure 2, A_{16} is a MRA at A_{19}. However, it is not a MRA at A_{13} since A_{13} appears before the A_{16}. Also, A_{13} is not a MRA at other two assertions since there exists a path that does not go through A_{13} but reaches to them (A_{16} and A_{19}).

2) MRAs Data Flow Formalization: We present the data flow formalization for MRAs computation at *procedure* level using forward information flow. It considers one run-time property at a time while computing MRAs, for example, MRAs for partitioning of ZD and AIOB assertions are computed separately. The equations below are shown for a node n in a *control flow graph* [13], where n denotes either an assignment statement or an expression controlling the flow.

$$In_n = \begin{cases} \emptyset & n \text{ is the entry node} \\ \bigcap_{p \ \epsilon \ \text{predecessors}(n)} Out_p & \text{otherwise} \end{cases} \quad (1)$$

$$Out_n = Gen_n + (In_n - Kill_n(In_n)) \quad (2)$$

$$Gen_n = \begin{cases} \{A\} & n \text{ has an assertion } A \\ \emptyset & \text{otherwise} \end{cases} \quad (3)$$

$$Kill_n(X) = \begin{cases} killInfo(X, n) & n \text{ modifies at least one variable} \\ \emptyset & \text{no variable is modified by } n \end{cases} \quad (4)$$

$$killInfo(X, n) = \{A \in X \mid (usedVars(A) \cap modifiedVars(n)) \neq \emptyset\} \quad (5)$$

$$usedVars(A) = \textit{r-values} \text{ from assertion } A \quad (6)$$

$$modifiedVars(n) = \textit{l-values} \text{ from program statement } n \quad (7)$$

In the above formalization,

- In_n represents the MRAs flowing in to n, i.e., at the start of n, whereas Out_n represents the MRAs flowing out (at the exit) of n.
- In_n equation (1) uses intersection because the flow information being computed is a *must* information.
- information flowing in at a point is computed using the information flowing out of its predecessors. This is because we compute the flow information in the forward direction.
- MRAs information is generated only at program points having assertions while kill information is computed at each variable modification point.
- In_n equation (1) indicates that MRAs are computed at procedure level. This equation will need a change if the MRAs are to be computed at the application level. The change is required to incorporate the effect of calling contexts and function call points.

3) Must Live Assertion (MLA): An assertion with expression e at a program point P is a MLA at its preceding program point P' if every path coming out of P' also passes through P and, no path segment between P' and P contains a *l-value* occurrence of any of the *r-value(s)* of e. This ensures each execution path having P' on it, also includes P.

In Figure 2, A_{16} is a MLA at A_{13}. However, it is not a MLA at A_{19} since A_{19} does not precede A_{16}. Also, A_{19} is not a MLA at the other two assertion points (A_{16} and A_{19}) since there exists a path that does not go through A_{19} but reaches them.

4) Data Flow Formalization for MLAs: The formalization for MLAs computation will be similar to that of MRAs. The only change here is the direction of information flow which is *backward* in case of MLAs. In order to account for this change, we change the In_n and Out_n equations as under. Out_n and In_n denote the MLAs being computed at the exit and start of a program point n respectively.

$$Out_n = \begin{cases} \emptyset & n \text{ is the exit node} \\ \bigcap_{s \ \epsilon \ \text{successors}(n)} In_s & \text{otherwise} \end{cases} \quad (8)$$

$$In_n = Gen_n + (Out_n - Kill_n(Out_n)) \quad (9)$$

5) Computation of LAs and FAs: Once MRAs and MLAs are available at each program point of an application, identification of partitions with their associated LAs becomes easy. We denote the MRAs *at the exit* of a program point n (flowing out of n) as $MRAs(n)$, and the MLAs *at its start* (flowing in at n) as $MLAs(n)$. Assertions A and A', with their assert expressions as e and e' respectively, form elements in the same partition if e and e' are structurally similar and one of the following hold:

1) $A \in (MRAs(A') \cup MLAs(A'))$. In this case, A' will be a FA and A can be its leader.
2) $A' \in (MRAs(A) \cup MLAs(A))$. In this case, A will be a FA and A' can be its leader.

An assertion A can be selected as a leader of a partition if for every other assertion A' in the partition, $A \in MRAs(A') \cup MLAs(A')$. If more than one assertion in a partition qualify to be a leader, one of them is randomly selected as the leader.

F. Assertions Partitioning: Limitation

```
11.      denom = ...;
12.      if(...){
13.          assert(denom != 0); r1 = n1/denom;
14.      } else {
15.          assert(denom != 0); r3 = n3/denom;
16.      }
```

Figure 3: Limitation scenario of PI-RAIT

The usage of MRAs and MLAs to identify LAs sometimes does not partition the assertions which are equivalent but not definitely reachable from one another. Examples of this are A_{13} and A_{15} in Figure 3. In spite of being equivalent, they will not be partitioned together because there will not be a MRA or a MLA available at an assertion point from another assertion.

III. PD-RAIT AND NVAIT

This section discusses the details of PD-RAIT and NVAIT, and presents an algorithm for both.

A. PD-RAIT

We have observed that, often, there are large number of assertions whose verification result follows the verification result of some other assertion, and they do not fall under the same partition during PI-RAIT. For instance, in Figure 1, A_{21} and A_{23} will not be identified as equivalent by PI-RAIT, even though the result of $V(A_{21})$ follows that of $V(A_{23})$. We use this characteristic peculiar to AIOB warnings to reduce the number of assertion verifications by partitioning them in a way similar to PI-RAIT.

1) Partitioning of AIOB assertions: In Figure 1, the need is to identify A_{23} as a leader and A_{21} as its follower. This will require backward analysis. Therefore, we use MLAs (as in PI-RAIT) to partition the AIOB assertions. The only change here is in the way MLAs are computed. The rest of the algorithm

to identify the LA and its associated FAs remains the same as in PI-RAIT.

We describe the change required in MLAs computation using the same example in Figure 1. We denote the MLAs at the exit of a program point *n* as $Out(n)$, and the MLAs at the start of *n* as $In(n)$. In PI-RAIT, $A_{23} \in Out(A_{21})$ but $A_{23} \notin In(A_{21})$ because A_{23} gets killed at A_{21}. This kill of A_{23} omits the association of A_{21} (follower) with A_{23} (leader). In PD-RAIT, we avoid such a kill computation. With this change, $A_{23} \in In(A_{21})$ and using PI-RAIT algorithm these will get partitioned together with A_{23} as the leader.

We skip the data flow formalization of MLAs computation in this technique due to lack of space. It is to note that this algorithm does not consider the MRAs for their partitioning. This approach can not be applied to partition the assertions associated with the AIOB warnings including a pre/post unary decrement operator or a signed data-type variable in their indexes. It is possible to refine the PD-RAIT technique to handle these limitation scenarios, but we do not do it as such scenarios are very rare in practice. Also, we do not apply this technique for ZD as its warnings with such patterns are very rare. Further, it can not be applied to partition overflow-underflow assertions since the relationship in verification results of LA and FAs can not be guaranteed.

```
  int *ptr1, **ptr2;
  #define NULL 0

    void f(...) {
      ...
11.   ptr1 = *ptr2;
12.
13.   assert(ptr1!=NULL); arr[0] = *ptr1++;
14.   assert(ptr1!=NULL); arr[1] = *ptr1++;
15.   assert(ptr1!=NULL); arr[2] = *ptr1++;
    }
```

Figure 4: DNP Assertions Example

Figure 4 presents another example for the Dereference of a Null Pointer (DNP) to illustrate the application of the PD-RAIT algorithm is property specific. The PD-RAIT algorithm used to partition the AIOB assertions can not be used for partitioning of these DNP assertions because successful verification of $V(A_{15})$ does not guarantee the same for $V(A_{13})$. However, it can be observed that if A_{13} is verified successfully, the successful verification of $V(A_{14})$ and $V(A_{15})$ is ensured. Thus, these assertions can be partitioned together with A_{13} tagged as a LA. It is intuitive that, such leader identification includes forward analysis, and hence, MRAs should be used instead of MLAs.

B. Non-Verifiable Assertions Identification Technique

Verification of an assertion by CBMC includes analysis of a provided entry function and the functions that are called directly or indirectly from the entry function. It uses a provided bound (unwinding count) for the unbounded loops during their unrolling. In the absence of this input (unwinding count) CBMC keeps unrolling the loop and eventually out of memory. When the input bound is not insufficient, it results into an unwinding assertion counterexample. This verification does not contribute in eliminating a false positive and is needless.

In practice, an application includes unbounded loops whose bound is determined only at the run-time. We present a few examples of the unbounded loops as follows:

1) an infinite loop such as $while(1)$, $for(;;;)$.
2) a loop in which a bound variable in its terminating condition takes values from library system calls.
3) a loop whose terminating condition is run-time dependent like $*ptr! =$'\0' and the string(s) pointed by ptr gets its content during run-time through $fgets(ptr)$.

If an assertion is *control* or *data dependent* on any of the above unbounded loops, then it is a NVA. An assertion A is *control* or *data dependent* on a loop l if A is dependent on a statement belonging to l. In Figure 1, A_{35} and A_{39} are the NVAs. This is because, each of them is *control* and *data dependent* on the unbounded loop starting at line 34. We skip the verification of these NVAs to make FPE faster.

We use the following algorithm to compute the NVAs:

1) Identify a set of unbounded loops (denoted as L_{UB}) used in the application. *Loop termination analysis* [14][15] can be used for this purpose.
2) For each assertion A and an entry function f_e,
   a) Identify the loops in f_e on which A is *control* or *data dependent*. Program dependence graph [16] can be used for this purpose. We denote this set of loops as L.
   b) If $(L \cap L_{UB}) \neq \emptyset$ then A is a NVA in the context of f_e.

It must be noted that the identification of an assertion as NVA is always with respect to an entry function. Further, this approach might wrongly mark an assertion as a NVA even if it is not. This, in turn, may make the FPE imprecise.

IV. Implementation & Experiments

This section covers the implementation details and experiments performed for the proposed FPE.

A. FPE Implementation

We implemented the proposed techniques in TCS Embedded Code Analyzer (TECA) [17] to eliminate false positives from the analysis warnings generated by it. TECA is a static analysis tool to verify C source code. We used the framework shown in Figure 5 to implement the proposed techniques. A short description of each of the component in the framework is provided below.

1) Static Analyzer: A static analysis tool (TECA) that performs verifications for properties AIOB, ZD, DNP, OFUF, etc. on input C code and reports safe, unsafe and warnings program points.

2) Code Annotator: This component annotates the source code to generate assertion corresponding to each warning produced by a static analysis tool.

 ISBN: 978-1-61208-307-0

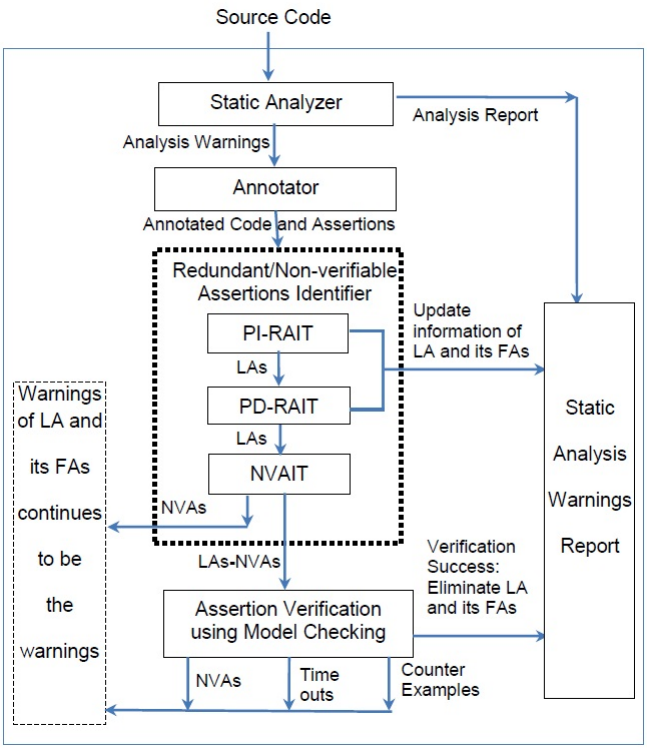

Figure 5: FPE Implementation Framework

3) Redundant/Non-Verifiable Assertions Identifier: We implemented the PI-RAIT, PD-RAIT and NVAIT as separate components, and use them in succession to get maximum benefit out of these techniques.

i. *PI-RAIT:* It implements the MRA and MLA formalizations in *context* and *flow* sensitive way at *function* level to partition the *input set* of assertions based on their equivalence. Further, it updates the analysis warnings report to reflect the association of LAs and their FAs.
ii. *PD-RAIT:* This component implements property specific PD-RAIT to partition the LAs computed by PI-RAIT. It does not analyze the FAs from PI-RAIT component since their analysis in this component would be redundant. Similar to PI-RAIT, it also updates the warnings report for its computed LAs and their FAs.
iii. *NVAIT:* It receives the PD-RAIT LAs and computes the NVAs from them. We used simple pattern based techniques to identify the unbounded loops in an application and did not use any complex loop termination analysis. In this component, due to lack of time, we used *must* reachability of unbounded loops instead of *control* or *data dependency*, to check if an assertion is a NVA.

4) Assertions Verifier: This component comprises mainly of a model checker (CBMC), and it optionally includes other tools implementing techniques such as code slicing [18], loops abstractions [9] to scale the model checker. The actual false positives elimination is performed by this component. It eliminates warnings (false positives) corresponding to a LA and its FAs when the LA is verified successfully. If the verification fails or times out, its corresponding warning is not eliminated. This component allows FPE in three different settings:

1) FPE at a *Function* level (F_{fpe}),
2) FPE at an *Application* level (A_{fpe}), *and*
3) FPE using a *Code context expansion* (C_{fpe}).

In C_{fpe}, assertions verifier component communicates with NVAIT component to check if the assertion being verified with an entry function is a NVA, and on finding the assertion as a NVA, its further verifications are skipped.

B. Experiments and Observations

We selected two embedded applications, one of 40 KLOC representing an automobile battery control module and another of 80 KLOC representing a smart card management system. Both the applications were verified for AIOB and ZD properties using abstract interpretation, and false positives were eliminated in three FPE settings - F_{fpe}, A_{fpe} and C_{fpe}. During our FPE experiments, we used-

1) 200 seconds time out for a CBMC verification.
2) sliced code with respect to generated assertion for its verification.
3) machine with Intel Core 2 Duo 2.33 GHz processor, 2 GB RAM configuration and having Windows XP SP3.

In Table I, we present details of the CBMC verification results for F_{fpe} and A_{fpe} settings without applying any of the proposed techniques (PI-RAIT, PD-RAIT and NVAIT). These results indicate that F_{fpe} outperforms A_{fpe} in terms of number of successful verifications. Also, there is a considerable number of unwinding counterexamples for these applications.

The results obtained for different combinations of the proposed techniques in each FPE setting are shown in Table II. For a FPE setting, it presents-

a. $|A_{in}|$, where A_{in} indicates a set of assertions those are verified by the CBMC.
b. $|E_{fp}|$, where E_{fp} is a set of false positives eliminated.
c. T_{fpe} representing the time taken ([Hours:Mins]) to verify assertions from A_{in}. It does not include the time spent in the code slicing.

In Table II, we also present the time taken in minutes (T_R) by a combination of the proposed techniques in a FPE setting. In our experiments, we applied PD-RAIT to AIOB only (and not for ZD). Following are the few observations from Table II.

1) In a setting, T_R is very less as compared to T_{fpe}, and this does not add any performance overhead in the FPE.
2) Among the FPE settings, the false positives eliminated are maximum in C_{fpe} and minimum in A_{fpe}.
3) Equivalent assertions found by PI-RAIT and PD-RAIT techniques, are more for AIOB compared to the ZD, and hence, the FPE time reduction is more for AIOB.
4) On an average, the PI-RAIT and PD-RAIT techniques have reduced 13.55% and 26.5% of FPE time respectively without compromising on the false positives eliminated. It indicates, although FPE with PI-RAIT and PD-RAIT is conservative in eliminating the false positives, in practice there is no miss on false positives eliminated.

 ISBN: 978-1-61208-307-0

TABLE I: Distribution of CBMC Verification Results

Application	Property	Setting	$\|A_{in}\|$	CBMC Timeouts	Verification Successful	CBMC Trace	Unwinding Assertions	CBMC Failures
Battery Control Module	AIOB	F_{fpe}	430	13	107	238	71	1
		A_{fpe}	430	0	1	0	429	0
	ZD	F_{fpe}	47	6	5	16	20	0
		A_{fpe}	47	0	0	0	47	0
Smart Card Management System	AIOB	F_{fpe}	314	5	13	231	65	0
		A_{fpe}	314	42	0	0	250	0
	ZD	F_{fpe}	54	0	24	22	8	0
		A_{fpe}	54	26	12	0	7	9

TABLE II: FPE Experiment Results

Application	Property	Techniques	T_R	F_{fpe}			A_{fpe}			C_{fpe}		
				$\|A_{in}\|$	$\|E_{fp}\|$	T_{fpe}	$\|A_{in}\|$	$\|E_{fp}\|$	T_{fpe}	$\|A_{in}\|$	$\|E_{fp}\|$	T_{fpe}
Battery Control Module	AIOB	-	-	430	107	04:11	430	1	04:14	430	270	13:45
		PI-RAIT	≈ 1	301	107	03:01	301	1	03:03	301	270	10:37
		PI-RAIT + PD-RAIT	≈ 1	196	107	01:50	196	1	01:53	196	270	06:47
		PI-RAIT + PD-RAIT + NVAIT	≈ 3	157	105	01:22	7	1	00:11	157	265	05:29
	ZD	-	-	47	5	01:06	47	0	00:29	47	8	02:16
		PI-RAIT	≈ 1	37	5	01:02	37	0	00:29	37	8	01:39
		PI-RAIT + NVAIT	≈ 2	22	2	00:16	0	0	00:00	22	5	00:41
Smart Card Management System	AIOB	-	-	314	13	01:45	314	0	09:38	314	22	29:13
		PI-RAIT	≈ 1	229	13	01:17	229	0	07:34	229	22	22:53
		PI-RAIT + PD-RAIT	≈ 2	177	13	01:03	177	0	06:43	177	22	19:14
		PI-RAIT + PD-RAIT + NVAIT	≈ 2	165	12	00:59	116	0	03:53	165	20	19:03
	ZD	-	-	54	24	00:14	54	12	00:41	54	29	01:28
		PI-RAIT	≈ 1	53	24	00:14	53	12	00:41	53	29	01:25
		PI-RAIT + NVAIT	≈ 2	50	24	00:13	26	10	00:24	50	29	01:24

5) On an average, the NVAIT technique has reduced the FPE time by 38.91% with the identification of 32.54% of assertions as NVAs, but it has compromised on 1.3% of false positives. This indicates that NVAIT makes FPE efficient as well as imprecise.
6) The application of all the three techniques, on an average, reduces the $|A_{in}|$ and T_{fpe}, respectively by -
   - 53.37% and 61% in F_{fpe} setting.
   - 82.37% and 71.4% in A_{fpe} setting.
   - 53.37% and 43% in C_{fpe} setting.
7) NVAIT technique when applied in A_{fpe} setting to battery control module, found all the assertions as the NVAs. The reason was traced to the inclusion of typical $while(1)$ loop implemented in *main* function of an embedded application.

V. Related Work

There are a number of techniques that combine static analysis with model checking to improve its precision. These techniques differ in a way these two are combined. Brat et al. [5] do this in such a way that static analysis component iteratively exchanges information with the model checker. The partial order information computed by static analysis is used by model checker for its state space reduction, and the aliasing information from model checking is used to refine the results of static analysis. Fehnker and Huuck [6] analyze the counterexamples generated through model checking by using abstract interpretation to learn new facts and refine the abstraction. This continues until a warning is either proved to be a bug or spurious.

Rödiger [19] combines data flow analysis and model checking to improve the security vulnerability detection. The vulnerable code statements are found based on invalidated user inputs and they are model checked to eliminate false positives or produce a readable counterexamples. Junker et al. [7] present an abstraction refinement technique to automatically find and eliminate the false positives. It is achieved by iteratively computing the infeasible sub-paths using SMT solvers and refining the models. Wang et al. [20] and Tsitovich [21] present techniques among the others that combine static analysis and model checking.

The techniques described above, combining static analysis and model checking, focus only on improving analysis precision. Further, these are difficult to use when static analysis is performed by widely used commercial tools (like Polyspace and Coverity) since it needs changes in the tool's back-end analysis. Post et al. [8] and Darke et al. [9] try to overcome this limitation by using model checking to eliminate the false positives produced by the static analysis tools. Of these two techniques, [8] presents an approach to eliminate the false positives generated by Polyspace, where it uses incremental context expansion to do so.

A major drawback of these two techniques ([8][9]) is that they generate and verify an assertion corresponding to each static analysis warning and, hence, involve numerous verification calls to a model checker. While we also generate an assertion corresponding to each of the output warning, we avoid verifying each assertion. We partition the generated

assertions and verify only one representative assertion from a partition, so that all the false positives in the partition are eliminated at once. We employ two techniques - one dependent on the property being verified and the other independent of it. Further, we try to skip a class of non-verifiable assertions before we pass it to a model checker.

VI. Conclusion and Future Work

In our experiments, we have found an abundance of redundant assertions in FPE. Our techniques helped in minimizing the verification calls to a model checker and, in turn, made the FPE faster. The property-dependent and property-independent RAITs marked 45% of the assertions as the *followers*. This is because there are multiple equivalent assertions in certain code regions and they often fall under the same partition. Eliminating the false positive corresponding to the leader of the partition eliminates all the false positives corresponding to the followers as well. This allows us to skip the verification of followers. Although we eliminated false positives conservatively in our approach, it was never the case in our experiments that we failed at eliminating one. The results of PD-RAIT technique indicate that using code-pattern based approach to partition assertions can be quite useful in reducing the FPE time. This is because these patterns are widely used.

The identification of NVAs based on unbounded loops, using NVAIT, is quite effective in minimizing the FPE time (led to an average reduction of 38.91%). There are many unbounded loops in an application and the need to verify an assertion dependent on them gets eliminated. This approach, being conservative, may find an assertion as a NVA even if it is not. That is to say, it might wrongly mark a verifiable assertion as a NVA that could have possibly eliminated a false positive. This explains the trade-off between precision and performance of FPE using this technique.

Our experiments depict that the reduction in FPE time is dependent on property being verified and the context at which false positives are eliminated. Although the experiments are performed on embedded domain applications written in C, we expect similar benefits on other domain applications as well due to common coding practices. These techniques can be extended further to verify properties in applications coded in other languages too.

We plan to experiment further with NVAIT technique replacing CBMC by SATABS [22]. We are also exploring a technique to make FPE more efficient by identifying assertions whose verifications are more likely to generate counterexamples.

References

[1] P. Cousot and R. Cousot, "Basic concepts of abstract interpretation," in *Building the Information Society*, ser. IFIP International Federation for Information Processing, R. Jacquart, Ed. Springer US, 2004, vol. 156, pp. 359–366. [Online]. Available: http://dx.doi.org/10.1007/978-1-4020-8157-6_27

[2] D. Engler, "Concur 2005 - concurrency theory," M. Abadi and L. de Alfaro, Eds. London, UK, UK: Springer-Verlag, 2005, ch. Static analysis versus model checking for bug finding, pp. 1–1. [Online]. Available: http://dx.doi.org/10.1007/11539452_1

[3] R. Jhala and R. Majumdar, "Software model checking," *ACM Comput. Surv.*, vol. 41, no. 4, 2009.

[4] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 27, no. 7, pp. 1165–1178, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2008.923410

[5] G. Brat and W. Visser, "Combining static analysis and model checking for software analysis," in *Proceedings of the 16th IEEE international conference on Automated software engineering*, ser. ASE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 262–. [Online]. Available: http://dl.acm.org/citation.cfm?id=872023.872568

[6] A. Fehnker and R. Huuck, "Model checking driven static analysis for the real world: designing and tuning large scale bug detection," *Innovations in Systems and Software Engineering*, vol. 9, no. 1, pp. 45–56, 2013. [Online]. Available: http://dx.doi.org/10.1007/s11334-012-0192-5

[7] M. Junker, R. Huuck, A. Fehnker, and A. Knapp, "Smt-based false positive elimination in static program analysis," in *ICFEM*, 2012, pp. 316–331.

[8] H. Post, C. Sinz, A. Kaiser, and T. Gorges, "Reducing false positives by combining abstract interpretation and bounded model checking," in *ASE*, 2008, pp. 188–197.

[9] P. Darke, M. Khanzode, A. Nair, U. Shrotri, and R. Venkatesh, "Precise analysis of large industry code," in *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, vol. 1, 2012, pp. 306–309.

[10] K. Vorobyov and P. Krishnan, "Comparing model checking and static program analysis: A case study in error detection approaches," in *International Workshop on Systems Software Verification (SSV'10)*, 2010.

[11] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.

[12] U. Khedker, A. Sanyal, and B. Sathe, *Data Flow Analysis: Theory and Practice*. Taylor & Francis, 2009. [Online]. Available: http://books.google.co.in/books?id=9PyrtgNBdg0C

[13] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, no. 7, pp. 1–19, Jul. 1970. [Online]. Available: http://doi.acm.org/10.1145/390013.808479

[14] A. Tsitovich, N. Sharygina, C. Wintersteiger, and D. Kroening, "Loop summarization and termination analysis," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, P. Abdulla and K. Leino, Eds. Springer Berlin Heidelberg, 2011, vol. 6605, pp. 81–95. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19835-9_9

[15] A. Bradley, Z. Manna, and H. Sipma, "Termination analysis of integer linear loops," in *CONCUR 2005 Concurrency Theory*, ser. Lecture Notes in Computer Science, M. Abadi and L. Alfaro, Eds. Springer Berlin Heidelberg, 2005, vol. 3653, pp. 488–502. [Online]. Available: http://dx.doi.org/10.1007/11539452_37

[16] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987. [Online]. Available: http://doi.acm.org/10.1145/24039.24041

[17] "TCS Embedded Code Analyzer (TECA)," [retrieved: October, 2013] http://www.tcs-trddc.com/trddc_website/scripts/project_detail.php?lab=SWRD&project_id=53,.

[18] A. D. Lucia, "Program slicing: Methods and applications," in *SCAM*, 2001, pp. 144–151.

[19] W. Rödiger, "Merging static analysis and model checking for improved security vulnerability detection," Masters, 2011. [Online]. Available: http://www.xn--wolfrdiger-icb.de/publication/roediger2011security.pdf

[20] L. Wang, Q. Zhang, and P. Zhao, "Automated detection of code vulnerabilities based on program analysis and model checking," in *SCAM*, 2008, pp. 165–173.

[21] in *Logic Programming*, ser. Lecture Notes in Computer Science, M. Garcia de la Banda and E. Pontelli, Eds., 2008, vol. 5366, pp. 822–823.

[22] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based predicate abstraction for ANSI-C," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, ser. Lecture Notes in Computer Science, vol. 3440. Springer Verlag, 2005, pp. 570–574.

 ISBN: 978-1-61208-307-0