Localizing Program Errors for Cimple Debugging

Samik Basu[1], Diptikalyan Saha[2], and Scott A. Smolka[2]

[1] Department of Computer Science, Iowa State University, Ames, IA 50014
E-mail: sbasu@cs.iastate.edu
[2] Department of Computer Science
State University of New York at Stony Brook, Stony Brook, NY 11794
E-mail: {dsaha,sas}@cs.sunysb.edu

Abstract. We present automated techniques for the explanation of counter-examples, where a counter-example should be understood as a sequence of program statements. Our approach is based on variable dependency analysis and is applicable to programs written in `Cimple`, an expressive subset of the C programming language. Central to our approach is the derivation of a focus-statement sequence (FSS) from a given counter-example: a subsequence of the counter-example containing only those program statements that directly or indirectly affect the variable valuation leading to the program error behind the counter-example. We develop a ranking procedure for FSSs where FSSs of higher rank are conceptually easier to understand and correct than those of lower rank. We also analyze constraints over uninitialized variables in order to localize program errors to specific program segments; this often allows the user to subsequently take appropriate debugging measures. We have implemented our techniques in the `FocusCheck` model checker, which efficiently checks for assertion violations in `Cimple` programs on a per-procedure basis. The practical utility of our approach is illustrated by its successful application to a fast, linear-time median identification algorithm commonly used in statistical analysis and in the Resolution Advisory module of the Traffic Collision Avoidance System.

1 Introduction

Model checking [22, 7] has recently made significant inroads in the domain of software verification [12, 18, 3, 11, 16, 4]. In this setting, model checking typically follows a three-step iterative process of abstraction, verification and refinement [24, 2, 17, 6]. First, given a system S, a finite-state abstraction S' of S is generated. Then, S' is verified with respect to the given property and a counter-example (sequence of program statements) is generated should a violation occur. Finally, S' is refined in case the counter-example is spurious (infeasible in S). The three steps are iterated until a feasible counter-example is identified or the abstract system satisfies the property.

In the event a feasible counter-example is generated, the user is left with the job of identifying the cause of the counter-example and taking appropriate corrective or debugging measures. However, the complex behavior of software systems, owing to the presence of complicated data and control structures, makes the process of decoding counter-examples extremely tedious, if not impossible.

```
1: int gotlock, lock;
2: bool error = false;
3: int main() {
4:  lock = 0;
5:  if (*) {
6:   while (*) {
7:    if (*) {
8:     getlock();
9:     gotlock++;
10:     bigProcedure();}
11:    if (gotlock == 1)
12:     rellock();}}

13: void getlock() {
14:  if (lock == 0)
15:   lock++;
16:  else error = true; }

17: void rellock() {
18:  if (lock == 1)
19:   lock–;
20:  else error = true; }

21: void bigProcedure() {
22:  . . . }
```

(a)

FSS$_1$ *Assumptions:* gotlock == 1

```
4:  lock=0              12:
5:                      18:  lock==1:false
6:                      20:  error=true
11:  gotlock==1:true
```

FSS$_2$ *Assumptions:* gotlock != 0

```
4:  lock=0              9:  gotlock++;
5:                      11:  gotlock==1:false
6:                      6:
7:                      7:
8:                      8:
14:  lock==0:true       14:  lock==0:false
15:  lock++             16:  error=true
```

(b)

Fig. 1. Simple locking program from [17].

To address this state of affairs, we present two automated techniques for effective error-reporting. The first of these is aimed at ranking counter-examples such that those counter-examples of higher rank are easier to understand and debug than those of lower rank. The second is a technique for localizing errors in programs to specific program regions, again allowing for effective identification and correction of program errors.

Our approach is based on variable dependency analysis and is applicable to programs written in `Cimple`, an expressive subset of the C programming language. Central to our approach is the notion of a *focus-statement sequence* (FSS), introduced by us in [5]. A FSS is a subsequence of a counter-example containing only those program statements that directly or indirectly effect the variable valuation leading to the program error behind the counter-example. Besides making counter-examples easier to understand by eliminating unnecessary details, FSSs can also be used to efficiently determine the feasibility of counter-examples, as the feasibility of the sequence of operations in an FSS implies the existence of a feasible counter-example.

Being based on variable dependency analysis, our approach also successfully identifies the constraints or *assumptions* over uninitialized or input variables necessary for the feasibility of a FSS. Such information can be used to understand program behavior in the context of different variable initializations. We have also developed the `FocusCheck` model checker for `Cimple`. It identifies all feasible counter-examples in a given `Cimple` program and presents these to the user in a precise and informative manner in terms of their FSSs and assumption sets.

Consider first our technique for ranking counter-examples. The basic idea behind this technique is to *rank* FSSs in terms of their length and the number of variables in their assumption sets. In [10], Engler and Ashcraft alluded to the importance of ranking counter-examples for easy inspection of deadlock errors in multi-threaded programs. In a similar vein, we order FSSs such that those of higher rank correspond to errors that are conceptually easier to understand and debug than those of lower rank.

Figure 1(a) presents a simple locking program correct behavior of which requires strict alternation between invocations of `getlock()` and `rellock()`; a violation occurs if `error=true` in any execution sequence of the program. The conditional construct `if(*)` is semantically equivalent to non-deterministic choice, required for representing conditional expressions whose variables are abstracted away to obtain finite data domain program. The `FocusCheck` model checker identifies two counter-

```
1: int main() {
2:  int x, y, z; bool error=false;
3:  int min=x, max=x;
4:  if (max < y) max=y;
5:  if (max < z) max=z;
6:  if (min > y) max=y;     ◁
7:  if (min > z) min=z;
8:  if !(min≤x ∧ min≤y ∧ min≤z ∧
         max≥x ∧ max≥y ∧ max≥z ∧)
9:     error=true;}
```

(a)

FSS_1	FSS_2	FSS_3
Assumptions:	*Assumptions*:	*Assumptions*:
x>y, x≤z	x>y, x>z, z>y	x>y, x>z, z≤y
3: min=x	3: min=x	3: min=x
6: min>y:true	6: min>y:true	6: min>y:true
6: max=y	6: max=y	6: max=y
7: min>z:false	7: min>z:true	7: min>z:true
8:	7: min=z	7: min=z;
9:	8:	8:
	9:	9:
Error:	*Error:*	*Error:*
max=y, min=x	max=y, min=z	max=y

(b)

Fig. 2. MinMax example from [13].

examples in terms of their FSSs and assumption sets (Figure 1(b)). The FSSs returned by `FocusCheck` are significantly smaller than the counter-examples from which they are derived as it discards program segments (e.g. the statements of `bigProcedure()`) that do not effect the valuation `error=true`. Variable dependency analysis tells us that in this case, the focus statements are those involving variables `gotlock`, `lock` and `error`. For each FSS returned by `FocusCheck`, the line numbers of the program statements in the FSS are given; the program statement itself is also given if it represents an operation on a variable of interest; e.g., statement `lock=0` at line 4 of FSS_1.

Our ranking procedure identifies the shorter of the two FSSs—FSS_1 comprising seven program statements, four of which are operations on the variables of interest—as the higher-ranked FSS. This directs the user to inspect FSS_1 before FSS_2. The user is also provided with the corresponding assumption set which reveals that the program behaves incorrectly if `gotlock` is initialized to 1; i.e., the error manifests when the condition at line 11 evaluates to true. Corrective measures therefore involve the appropriate initialization of `gotlock` (negate the assumption) in one of the statements leading to the statement at line 11. Ranking FSSs and their assumption sets narrows down the erroneous region in the program under investigation, and assist the user in taking corrective measure by inserting the assignment `gotlock=0` after line 5 or line 6.

Consider next our technique for localizing errors to specific program regions. This technique is based on the observation that in many practical scenarios, a single error in a program can lead to multiple counter-examples, owing to the program's branching behavior. Given a set of FSS, we generate a *reduced set of focus-statement sequences* by discarding the differences and analyzing the commonalities among the FSSs.

To illustrate our technique for localizing errors in programs, consider the `Cimple` program of Figure 2(a). The program is intended to compute the minimum and maximum of three integer variables but contains an obvious typo at line 6(marked in the figure): the assignment `max=y` should instead be `min=y`. The error condition is satisfied if the minimum or the maximum is set to an incorrect value. `FocusCheck` produces three FSSs corresponding to the three possible erroneous program behaviors (Figure 2(b)).

Our technique for localizing program errors is based on the elimination of a constraint and its negation, should they appear in two different assumption sets; it is easily shown that such a pair of constraints is irrelevant to the cause of the error. In the example, our technique first eliminates the constraints `z>y` and `z≤y` from the assumption sets of FSS_2 and FSS_3, respectively. FSS_3 is then discarded, being now identical to

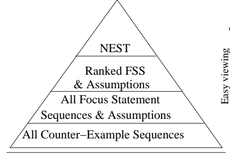

Fig. 3. Viewing counter-examples at different levels of detail.

FSS_2 in terms of its assumption set and line numbers. FSS_1 and FSS_2 contain x≤z and x>z, and we delete these constraints from their respective assumption sets. The remaining constraint in the assumption sets of FSS_1 and FSS_2 is x>y. We project x>y on both the FSSs and identify the if-block at line 6 as the region containing the error.

An important aspect of our technique for generating FSSs is that it proceeds in a modular fashion, handling each procedure in the program independently of the others. Specifi cally, our technique seeks to minimize the overhead of analyzing a procedure if it has been invoked from multiple call sites and each of these call sites are present in the counter-example sequence. Central to our technique is the summarization of each procedure with respect to the valuation of global variables.

Our techniques for ranking FSSs and for localizing program errors allow the user to view counter-examples at different levels of granularity and detail (Figure 3). At the lowest level, the user is presented with the entire counter-example sequence. At the intermediate levels, the user sees the FSS of the counter-examples, ordered in terms of their relative complexity. At the highest level, reduced sets of focus statements are identifi ed on the basis of the constraints in their assumption sets. As one moves to higher levels, information is organized and/or minimized without compromising its usefulness.

Contributions and organization of the paper. In summary, the main contributions of the paper may be seen as follows:

1. We present a hierarchy of automated techniques aimed at allowing users to effectively ascertain the root cause of a program error. To the best of our knowledge, this is the fi rst effort to organize error explanation at different levels of granularity.
2. Focus-statement sequences, introduced in [5] and reviewed in Section 2, use variable dependency analysis to make counter-examples much easier to comprehend by discarding unnecessary details. We introduce in Section 3.1 a procedure for ranking FSSs such that FSSs higher in the ranking correspond to errors that are conceptually easier to understand and debug than those lower in the ranking.
3. Our technique for generating a reduced set of FSS from a given set of FSSs and their assumptions proceeds by discarding the differences and analyzing the commonalities among the given FSSs (Section 3.2). It can signifi cantly aid the user in localizing the region within a program containing the error under investigation.
4. We have implemented our error-localization technique in the `FocusCheck` model checker. At its core, the model checker performs reachability analysis of programs to generate all possible feasible counter-examples in terms of their FSSs (Section 4). Reachability analysis is performed in a modular fashion by summarizing the effects of a given procedure independently of all other procedures (Section 3.3).
5. We demonstrate the effectiveness of our technique by analyzing the resolution advisory module (RA) of the traffi c collision avoidance system (TCAS) (Section 5).

(a)

```
1: int gotlock, lock;
2: void main() {
3:  int old, new;
4:  lock = 0;
5:  if (*) {
6:   while (*) {
7:    gotlock = 0;
8:    if (*) {
9:     getlock();
10:    gotlock = gotlock + 1;
11:   }
12:   if (gotlock == 1)
13:    rellock();
14:  }
15: }
16: lock = 0;
17: bigProcedure();
18: while (new!=old) {
19:  getlock();
20:  new = old;
21:  if (*) {
22:   rellock();
23:   new = new + 1;
24:  }
25: }
26: rellock();
27: return;
28: }
29: void getlock() {
30:  bool error = false;
31:  if (lock == 0)
32:   lock = 1;
33:  else error = true;
34:  return;
35: }
36: void rellock() {
37:  bool error = false;
38:  if (lock == 1)
39:   lock = 0;
40:  else error = true;
41:  return;
42: }
43: void bigProcedure() {. . .}
```

(b)

Error Condition	Counter-Example	Focus Statement Sequence
error = true in rellock()	$\langle$4, 5, 6, 7, 8, 9, 30, 31, 32, 34, 10, 12, 13, 37, 38, 39, 41, 16, 17, . . ., 18, 26, 37, 38, 40$\rangle$	$\langle$16, 18, 26, 38, 40$\rangle$
error = true in getlock()	$\langle$4, 5, 6, 7, 8, 9, 30, 31, 32, 34, 10, 12, 13, 37, 38, 39, 41, 16, 17, . . ., 18, 19, 30, 31, 32, 34, 20, 21, 18, 19, 30, 31, 33$\rangle$	$\langle$16, 18, 19, 31, 32, 34, 20, 18, 19, 31, 33$\rangle$

Fig. 4. (a) The locking example (expanded version). (b) Counter-Examples and FSS.

2 Preliminaries

In this section, we provide a brief overview of our technique for extracting focus-statement sequences from counter-examples [5]. Given a program and a correctness assertion, a *counter-example* is a sequence of statements executed by the program leading to a violation of the assertion. A *focus-statement sequence* (FSS) is a subsequence of a counter-example such that each statement in the subsequence directly or indirectly effects the variable valuations responsible for the assertion violation in the program.

Focus statement sequence: slicing counter-example. Our technique of identifying FSSs, which are semantically dependent, possibly noncontiguous program segments, is based on *program slicing* [9, 19]. Counter-examples are generated during model checking via reachability analysis from the start state of the program to a state violating the correctness condition. Reachability analysis is also used to record the dynamic control and data dependencies at each statement. Note that the last statement of a counter-example is responsible for the violation of correctness condition. A statement in a counter-example is classifi ed as focus statement if it directly or indirectly effects the last statement in the counter-example. In short, FSS is identifi ed by slicing counter-example using the last statement in the counter-exmaple as the slicing criterion.

Definition 1 (Focus Statement). *Given a counter-example sequence* $S = \langle s_1, s_2, \ldots, s_n \rangle$ *where* s_i *denotes the* i*-th program statement along with its line number and* s_n *is the last statement in the counter-example,* s_j *is said to be a focus statement if any one of the following holds:*

1. s_j *is in the dynamic slice of S w.r.t. slicing criterion* s_n

2. *s_j is a call or return statement with at least one focus statement in the body of the called procedure.*

Here we describe the exact slicing method we employ for extraction of FSSs. We start the backward exploration from the last statement of a counter-example. A set `Error` is maintained during the analysis, containing the line numbers of the statements and variables that effect the last statement. `Error` is generated from the sets `control(s)` and `data(s)` of each statement `s` representing control and data dependencies at a statement `s` respectively. A statement `s` is control dependent on those conditional statements whose line numbers are present in `control(s)`, while `s` depends on the variables in `data(s)`.

For the first counter-example given above, the set `Error` is initialized to $\{38\}$, as the last statement at line 40 of the counter-example has $\texttt{control}(40) = \{38\}$. As backward analysis proceeds, the statement at line 38 is encountered with $\texttt{Error} = \{38\}$; therefore, the statement at line 38 is classified as a focus statement and `Error` is updated by removing 38 and introducing the variable `lock` ($\texttt{lock} \in \texttt{data}(38)$). The next statement reached in the backward exploration is the one at line 37 (`error=true`). However, the variable $\texttt{error} \notin \texttt{Error}$. Therefore the statement at line 37 is not a focus statement and the `Error` set remains unaltered. Backward exploration terminates if the `Error` set is empty or all the statements in the counter-example have been analyzed. The focus-statement sequences thus identified for each of the two counter-examples given in Figure 4(b).

Feasibility of counter-example sequences. The behavior of a program typically depends upon the valuation of variables that are inputs to the program. If an input variable has an infinite domain such as an integer variable, reachability is performed by leaving the operations on these variables uninterpreted. For example, the operations at lines 18, 20 and 23 in Figure 4 are uninterpreted and forward reachability is performed by considering all possible (boolean) valuations of the conditional expression at line 18.

This assumption leads the model checker to consider both the branches in the conditional expression, whereas in reality only one of the branches is feasible. This results to infeasible counter-examples in the output of the model checker. Typically, feasibility analysis involves considering each counter-example to determine whether all the operations in the counter-example are consistent in the original source program. In contrast, we reduce the overheads of feasibility analysis by reducing (a) number of counter-examples to be checked and (b) number of operations to be checked for consistency.

These reductions are achieved from the observation that, given an FSS F with feasible sequence of operations, there exists at least one feasible counter-example C with F as its subsequence. To check the feasibility of counter-examples we check the feasibility of the sequences of operations in the counter-examples' FSSs. Note that the length of a FSS is often less than that of the corresponding counter-example making feasibility checking of the former more efficient than the latter. Furthermore, if multiple counter-examples correspond to the same FSS, proving/disproving feasibility of all these counter-examples is performed by checking for feasibility of a single FSS.

In the above example, the second FSS is infeasible due to the infeasibility of the operation at line 20 followed by the operation at line 18. The first FSS, however, is

feasible. FSSs can, therefore, be used to both shorten counter-examples by eliminating unnecessary details and to effectively discard infeasible execution sequences.

3 Debugging Cimple Programs

3.1 Ranking the Counter-Examples

In many practical settings, a single error can force a program to behave erroneously in multiple ways leading to the generation of multiple counter-example sequences. For example, a single error in each of the programs of Figures 1 and 2 (missing initialization and incorrect assignment, respectively) generates more than one counter-example. While it is diffi cult, if not impossible, to localize the cause of multiple counter-examples to one single parameter in a program, it is possible develop techniques that will effectively guide the users toward making the correct choice in debugging programs. Ranking counter-examples on the basis of focus-statement sequences and their assumption sets is a methodology where error traces are sorted in terms of their complexity.

Definition 2 (Rank). *Given two FSSs F_1 and F_2, F_1 is said to be of higher rank than F_2, denoted by $F_1 \succeq F_2$, if:*

1. *the length of F_1 is less than that of F_2 or*
2. *the length of F_1 is equal to the length of F_2 and the number of variables in the assumptions of F_1 is less than number of variables in the assumptions of F_2.*

Defi nition 2 defi nes a partial order over FSSs. Higher ranked FSSs are more likely to be easier to parse and understand than the lower ranked ones. The rationale for selecting the two ranking criteria is based on the following observations. A user can potentially parse a smaller sequence of focus statements than a longer one. If two FSSs involve an identical number of statements, the one which requires assumptions over a fewer number of variables for its feasibility is potentially simpler to understand than the one requiring constraints over more variables. The involvement of a fewer number of variables in an assumption set means that a fewer number of program variables are effected by the assumptions and, therefore, the user will be required to concentrate on a fewer number of operations on variables in order to track down the error.

3.2 Localizing Program Errors using Assumption Sets

In this section, we provide a methodology to further minimize the sequence of statements in each FSS that the user is required to inspect in order to fi nd the potential cause of an error. Our objective is to localize the program error to a specifi c segment of an FSS. We show that in certain scenarios, our technique can identify the exact program statement that is the cause of the error, and provide useful feedback to the user about a possible remedy.

Our approach is based on algorithm Reduce given in Figure 5; it performs a reduction on a given set of FSS-assumption set pairs followed by a projection of the resulting assumption sets to their corresponding FSSs.

Input: A set S of FSSs $F_1, F_2, \ldots, F_n$ and their corresponding assumption sets $A_1, A_2, \ldots, A_n$.
Output: Reduce(S).

1. Initially Reduce(S) = S. Repeat Steps 2 and 3 till no change in Reduce(S).
2. If there exits a constraint c in a unique A_i and its negation $\neg c$ in a unique A_j, $i \neq j$, then delete c from A_i and $\neg c$ from A_j. Iterate this step until no such c is found.
(*reduction by eliminating complementary assumptions*)
3. If there exist in Reduce(S) identical FSSs F_i and F_j with identical assumption sets A_i and A_j, $i \neq j$, remove any one of these FSS-assumption set pairs from Reduce(S).
(*reduction by eliminating identical FSS-assumption pairs*)
4. Project each A_i in Reduce(S) to its corresponding FSS F_i as follows.
5. Start with statement s_k, $k = 1$, the first statement in F_i and repeat the following cases.
   (a) s_k is a conditional statement with conditional expression c:
      i. if $c \notin A_i$ or $\neg c \notin A_i$ then mark all focus statements in the block containing s_k and go to step 4
      ii. else k++
   (b) s_k is an assignment statement $x = y$ (*call statements are considered as assignments of actual parameters to the corresponding formal parameters*)
      i. if $\exists c \in A_i$ involving y then add the new constraint over x in A_i by replicating constraints over y and replacing y by x in the replication. k++
      ii. if $\exists c \in A_i$ involving x then delete c from A_i. k++
   (c) If s_k is the last statement of F_i, mark the entire F_i; go to step 4.

Fig. 5. Algorithm Reduce.

Definition 3 (Reduced Set of Focus Statement Sequences). *A set of focus-statement sequences $\{F_1, F_2, \ldots, F_n\}$, where each F_i is paired with assumption set A_i, is said to be* reduced *if the following conditions hold:*

1. *If c is a constraint in A_i, then $\neg c$ is either not present in any A_j or is present in at least two A_j ($j \neq i$) (reduction of assumptions).*
2. $\forall i, j (i \neq j) \Rightarrow (F_i \neq F_j \vee A_i \neq A_j)$ *(reduction of FSSs).*
3. *A sequence of statements $\langle s_{i_1}, s_{i_2}, \ldots, s_{i_n} \rangle$ is* marked *in each FSS F_i such that the outer-most conditional expression[1] over input variables in F_i cannot be evaluated using the constraints present in A_i (Neighborhood of Error Statements).*

Eliminating complementary assumptions. Recall that assumption sets represent the constraints on uninitialized or input variables necessary for the feasibility of a counter-example sequence. Specifi cally, assumptions represent the constraints necessary to validate the conditional expressions present in the counter-example. Our technique for eliminating complementary assumptions from a set of FSSs is based on the following observation:

[1] A conditional expression is outer-most in an FSS if it appears in the first conditional statement in the FSS

If a constraint c *and its negation* $\neg c$ *appear in* exactly two *distinct assumption sets, then* c *and* $\neg c$ *are most likely generated from the same conditional statement which has exactly one FSS for each of its branches.*

There are three possible ways in which the above observation holds:

(a) **Error statement followed by a conditional.** Consider first the case where an error in a program is caused by an incorrect assignment that is followed by a conditional block. If the assignment affects statements in both branches of the conditional block and if these statements affect the assertion violation, then two FSSs are generated. Each FSS is accompanied by an assumption set containing the constraint required to obtain the appropriate valuation of the conditional expression; i.e., a constraint and its negation appear in two different assumption sets.

(b) **Error in a conditional expression.** An error caused by an incorrect conditional expression also produces at least two FSSs. This is due to the fact that both branches in the conditional lead to the assertion violation. Thus, each branch leads to the generation of an FSS along with an assumption set containing the constraint required for the corresponding valuation of the conditional expression.

(c) **Errors in both branches of a conditional block.** This case corresponds to the situation where there are errors in both branches of a conditional block.

In all of the above cases, the pair under consideration (a constraint c and its negation $\neg c$ appearing in two different assumption sets) can be safely classified as uninteresting constraints. The reason for this is that negating the conditional expression in the program that generated c and $\neg c$ followed by reachability analysis of error state will generate the same set of FSSs. In case the pair of constraints is generated from two different conditional statements our method will localize the bug in an ancestor block of the block containing the error. This imprecision can be removed by associating program location with each assumptions and eliminating complimentary assumption only if they are generated from the same program location.

Another important feature of the constraint pair is that they must appear in exactly two assumption sets. The requirement of exactly two instead of at least two assumption sets has its root in the following observation. Suppose there are two assumption sets containing constraint c and a single assumption set containing $\neg c$; i.e., there are two FSSs corresponding to constraint c and one FSS corresponding to $\neg c$. In this case, it is most likely there are errors present in both branches of a conditional block with conditional expression c (or $\neg c$) (see above). Further, as there are multiple FSSs corresponding to c, the error is present in a block nested in the then-branch or else-branch of the conditional. Our aim is to localize the error in the block nested inside the conditional statement. As such, in this situation, we do not discard the constraints c and $\neg c$.

Removal of a constraint and its negation from two assumption sets might make these FSSs and their corresponding assumption sets identical; one of of these FSSs can be safely removed from further consideration. Reduction is therefore achieved in two different dimensions: the size of assumption sets and the number of FSSs. Steps 1–3 of algorithm Reduce encode the reduction steps described in this section. The following section describes our technique for identifying erroneous program segments in each of the remaining FSSs.

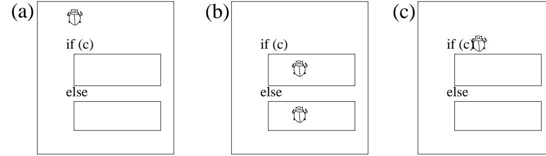

Fig. 6. NESTs are the focus statements in the outer and inner blocks for different bug positions.

Projecting assumptions to focus-Statement sequences. Our technique is based on projecting the assumptions (left after discarding complementary constraints) on the corresponding FSS. We refer to the resulting subsequence as the Neighborhood of Error STatements (NEST), the region in the FSS where the user must apply corrective measures to remedy the corresponding counter-example (steps 5(a), (b) and (c) in Figure 5).

Projection proceeds by forward analysis of the FSS. Each statement may or may not update the assumption set depending on whether or not it affects the constraints in the assumption set. Each statement is interpreted under the assumption set obtained after analyzing the statement preceding it in the FSS. The first statement is interpreted using the reduced assumption set of the FSS obtained by discarding complementary constraints in the assumption sets.

The terminating condition (5(a)i in Figure 5) implies that we have identified the outermost conditional statement whose condition has generated unimportant constraints (discarded in the previous steps of the algorithm). Next, we mark the NEST as all of the focus statements that belong to the same block [2] as the conditional statement (error localization). Note that the size of the NEST can be significantly smaller than the actual program block in which it belongs as only the statements responsible for the assertion violation (i.e., the subsequence of the FSS) are included in the NEST.

The NEST presents to the user a region which encompasses the error statement(s) present in the program (See figure 6). In the worst case (e.g., only one FSS is generated due to the program error), the NEST encompasses the entire FSS, while in the best case, NEST identifies the exact statement which, if altered, will remove the program error.

Analyzing the median identification program. We illustrate the effectiveness of algorithm Reduce using the program given in Figure 7. This program sorts five integers `a1`, `a2`, `a3`, `a4`, and `a5`, in only five comparisons given the partial order `a1>a2, a3>a4, a1>a3`. The output of the program is a sorted list of output variables `o1`, `o2`, `o3`, `o4`, `o5` in descending order. The program is based on the algorithm for finding the median of a list of numbers in linear time [8].

The program proceeds by considering inequalities between pairs of inputs. Consider first the two cases when `a3>a5` [lines 9-40] and `a3≤a5` [lines 41-61]. In the former case, if `a4>a5` is satisfied, the program proceeds to identify the correct position for `a2` in the ordered list `a3>a4>a5` [lines 12-24]. A similar technique is used for `a4≤a5` when the ordering is `a3>a5>a4` [lines 26-38]. In the latter case, i.e., when `a3≤a5`, the condition `a2>a3` is used to find the correct position of `a5` with respect to the ordering `a1,a2,a3` [lines 42-50]; on the other hand, `a2≤a3` implies that the sorted ordering is `a1`, `a5`, `a3`, `a2`, `a4` [lines 52-61].

[2] A block refers to all of the statements which are in the same or nested static scope.

```
1: int main(){
2: int a1,a2,a3,a4,a5;
3: int o1,o2,o3,o4,o5;
4: int error=0;
5: // input a1, a2, a3, a4, a5
6: if(!((a1>a2)&&(a3>a4)
       &&(a1>a3))){
7:     exit(0);
8: }
9:  if(a3 > a5){
10:   o1=a1;
11:   if(a4 > a5)
12:     if(a2 > a4){
13:       o4=a4,o5=a5;
14:         if(a2 > a3)
15:           o2=a2,o3=a3;
16:         else
17:           o2=a3,o3=a2;
18:       }else{
19:         o2=a3,o3=a4;
20:         if(a2 > a5)
21:           o4=a2,o5=a5;
22:         else
23:           o4=a5,o5=a2;
24:       }
25:     else /* line 11 */
26:       if(a2 > a5){
27:         o4=a5,o5=a4;
28:         if(a2 > a3)
29:           o2=a2,o3=a3;
30:         else
31:           o2=a3,o3=a2;
32:       }else{
33:         o2=a3,o3=a5;
34:         if(a2 > a4)
35:           o4=a2,o5=a4;
36:         else
37:           o4=a4,o5=a2;
38:       }
39:   }
40:   else
41:     if(a2 > a3){
42:       o4=a3,o5=a4;
43:       if(a5 > a2){
44:         o3=a2;
45:         if(a5 > a1)
46:           o1=a5,o2=a1;
47:         else
48:           o1=a1,o2=a5;
49:       }else
50:         o1=a1,o2=a2,o3=a5;
51:     }else{
52:       o3=a3;
53:       if(a1 > a5)
54:         o1=a1,o2=a5;
55:       else
56:         o1=a5,o2=a1;
57:       if(a2 > a4)
58:         o4=a2,o5=a4;
59:       else
60:         o4=a4,o5=a2;
61:     }
62:   if((o1<o2)||(o2<o3)||
      (o3<o4)||(o4<o5)){
63:     error=1;
64:   }
```

Fig. 7. Sorting five partially ordered numbers.

Consider now an error in the program caused by an artificially injected incorrect conditional expression at line 14: $\texttt{a2} < \texttt{a3}$ instead of $\texttt{a2} > \texttt{a3}$. In this case we will get two FSSs, $F_1 = \langle 4, 6, 9, 10, 11, 12, 13, 14, 15, 62, 63 \rangle$ with assumption set A_1= $\{\texttt{a1} > \texttt{a2}, \texttt{a3} > \texttt{a4}, \texttt{a1} > \texttt{a3}, \texttt{a3} > \texttt{a5}, \texttt{a4} > \texttt{a5}, \texttt{a2} > \texttt{a4}, \texttt{a2} > \texttt{a3}\}$ and $F_2 = \langle 4, 6, 9, 10, 11, 12, 13, 14, 17, 62, 63 \rangle$ with assumption set A_2= $\{\texttt{a1} > \texttt{a2}, \texttt{a3} > \texttt{a4}, \texttt{a1} > \texttt{a3}, \texttt{a3} > \texttt{a5}, \texttt{a4} > \texttt{a5}, \texttt{a2} > \texttt{a4}, \texttt{a2} \leq \texttt{a3}\}$. Step 2 of algorithm Reduce will delete the constraints $\texttt{a2} > \texttt{a3}, \texttt{a2} \leq \texttt{a3}$ from assumption sets A_1 and A_2, respectively. In steps 4 and 5 of the algorithm we project the modified set A_1 on F_1 and modified set A_2 on F_2. NEST is identified as $\langle 13, 14, 15 \rangle$ for F_1 and $\langle 13, 14, 17 \rangle$ for F_2.

Introducing another bug at line 48 by copying line 46 to line 48 will generate three FSSs, two of which are the same as F_1 and F_2 described above. The third FSS F_3 comprises ($\langle 4, 6, 9, 40, 41, 42, 43, 44, 45, 48, 62, 63 \rangle$) and its corresponding assumption set A_3 is $\{\texttt{a3} \leq \texttt{a5}, \texttt{a2} > \texttt{a3}, \texttt{a5}, > \texttt{a2}, \texttt{a5} \leq \texttt{a1}\}$. In this case, step 2 of our algorithm will not delete the constraint $\texttt{a3} > \texttt{a5}$ and $\texttt{a3} \leq \texttt{a5}$ from any assumption sets as $\texttt{a3} > \texttt{a5}$ exists both in A_1 and A_2. This will led us to identify NESTs $\langle 3, 14, 15 \rangle$ and $\langle 13, 14, 17 \rangle$ as in the previous case. This justifies the deletion of a constraint and its negation only if they exist in exactly two distinct FSSs. In the present case, there exists only one FSS (F_3) that goes through the else-block of the condition at line 9. We cannot localize the error for F_3 since step 5 of algorithm Reduce is iteratively executed until we reach the last statement in F_3 and as such the entire F_3 is marked as a NEST.

3.3 Detecting Focus-Statement Sequence Modularly via Summarization

Model checking involves finding whether an error state in the system is reachable from its start state. Efficient reachability analysis [11, 21] of programs with recursions employs summarization of procedures with respect to the valuation of global variables. Intuitively, summarization represents the effect of a procedure and involves computing the relation between variable valuations at its start and exit points. The main advantage

of this technique is modularity and efficiency; each procedure is analyzed in isolation and their summaries are used for forward reachability analysis. Observe that program behavior is classified using three types of transitions:

Statement	Transition	Stack depth	Global variable valuations
return:	$s, g \longrightarrow \epsilon, g'$	decreases by 1	g and g' respectively before and after executing s
call:	$s, g \longrightarrow s_1, g_1 : s_2, g_2$	increases by 1	g, g_1, g_2 at call statement s of callee, start statement s_1 of the called procedure and return location s_2 of the callee respectively
other:	$s, g \longrightarrow s', g'$	no change	g and g' respectively before and after executing s

Based on the above observations, the effect of a procedure on the global variables is the least fixed point of relation $\texttt{fsum}(g, s, g')$, where s is the start state of the procedure and g and g' are the valuations of global variables at the entry and exit point of the procedure, respectively. It can be shown that the procedure with m transitions can be summarized in time $O(m \times g^3)$ [11].

$$\begin{array}{lcl} \texttt{fsum}(g, s, g') & \Leftarrow & s, g \longrightarrow \epsilon, g' \\ \texttt{fsum}(g, s, g') & \Leftarrow & s, g \longrightarrow s_1, g_1 : s_2, g_2 \wedge \texttt{fsum}(g_1, s_1, g_2) \wedge \texttt{fsum}(g_2, s_2, g') \\ \texttt{fsum}(g, s, g') & \Leftarrow & s, g \longrightarrow s_1, g_1 \wedge \texttt{fsum}(g_1, s_1, g') \end{array}$$

To find a sequence of statements leading to violating state our `FocusCheck` model checker performs forward reachability analysis from the start state of the program. Each call site in the program is interpreted in terms of the effect of the called procedure on the global variables. In other words, if there are multiple calls (say k) to the same procedure, the called procedure is analyzed *once* to compute the `fsum` relation instead of analyzing it k times. Summarization, therefore, makes a significant contribution to the efficiency of model checker.

Summarizing effects of procedures using backward reachability. Focus statements are identified by backward reachability analysis of counter-examples (Section 2). The technique involves dynamically computing a set `Error` consisting of variables whose valuations directly or indirectly effect the variable valuation that caused assertion violation. As in forward reachability analysis, backward reachability is also performed efficiently using summarization of procedures. Given the set `Error` and the valuations of global variables at the exit point of a procedure, summarization involves computing the `Error` set, the valuation of global variables at the start location of the procedure, and the sequences of focus statements. The summary of a procedure computed via backward analysis is defined by the least model of the $\texttt{bsum}(g, e, \texttt{fss}, s, g', e', \texttt{fss}')$ relation, where (a) s is the start location of the procedure, (b) $g', e', \texttt{fss}'$ are the valuation of the global variables, the `Error` set, and the sequence of focus statements at the exit point of the procedure and (c) $g, e, \texttt{fss}$ are the valuation of global variables, the `Error` set, and sequence of focus statements at the entry point.

$$\begin{array}{lcl} \texttt{bsum}(g, e, \texttt{fss}, s, g', e', \texttt{fss}') & \Leftarrow & s, g \longrightarrow \epsilon, g' \wedge \texttt{update}(e, \texttt{fss}, s, e', \texttt{fss}') \\ \texttt{bsum}(g, e, \texttt{fss}, s, g', e', \texttt{fss}') & \Leftarrow & s, g \longrightarrow s_1, g_1 : s_2, g_2 \wedge \texttt{bsum}(g_2, e_2, \texttt{fss}_2, s_2, g', e', \texttt{fss}') \wedge \\ & & \texttt{bsum}(g_1, e_1, \texttt{fss}_1, s_1, g_2, e_2, \texttt{fss}_2) \wedge \texttt{update}(e, \texttt{fss}, s, e_1, \texttt{fss}_1) \\ \texttt{bsum}(g, e, \texttt{fss}, s, g', e', \texttt{fss}') & \Leftarrow & s, g \longrightarrow s_1, g_1 \wedge \texttt{bsum}(g_1, e_1, \texttt{fss}_1, s_1, g', e', \texttt{fss}') \wedge \\ & & \texttt{update}(e, \texttt{fss}, s, e', \texttt{fss}') \end{array}$$

$\texttt{update}(e, \texttt{fss}, s, e', \texttt{fss}')$ checks whether the statement s effects the `Error` set (e'). $\texttt{fss}'$ is the FSS identified up to the point statement s is visited in backward reachability analysis. In the event s is classified as a focus statement, `fss` is generated by prepending s to $\texttt{fss}'$ while e' is appropriately updated to e.

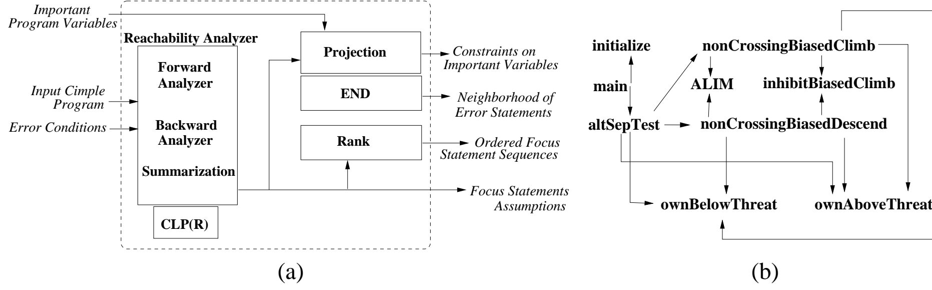

Fig. 8. (a) Architecture of the `FocusCheck` model checker. (b) Call graph for RA module.

The distinguishing feature between `fsum` and `bsum` relations, used for summarizing procedures during forward and backward reachability analysis respectively, is the order in which the transition between statements appearing in the execution sequence is analyzed. Consider the second rule in the defi nition of the relations `fsum` and `bsum`, the case where s corresponds to a call statement. The `fsum` relation proceeds by computing the `fsum` of the called procedure followed by the `fsum` of the callee starting from the return location. On the other hand, `bsum` fi rst computes the `bsum` of the callee starting from the return location followed by the `bsum` of the called procedure.

However, the common aspect of `fsum` and `bsum` is that summarization makes forward and backward analysis of program traces effi cient. Both relations once computed for each procedure are used multiple times if the same procedure is invoked multiple times in a program with the same input/output parameters (global valuations, `Error` sets, focus statements). To illustrate the impact of the `bsum` relation in fi nding FSSs, consider the following example. Assume procedure Q is invoked k times in the error trace in procedure P, and Q has m different paths from its start to exit point. Further assume that Q does not effect the counter-example sequence and as such does not contribute to set `Error`. That is, statements in Q do not appear in the FSS. Naive backward reachability analysis from the error state will analyze Q by inlining the procedure at its call sites; i.e., the m different paths in Q will be analyzed k times. On the other hand, summarization will analyze the m different paths in Q only once and use the summary result at each of the call sites.

4 Tool Description

In this section, we describe the salient features of our `FocusCheck` model checker `FocusCheck`, in which we have implemented the techniques described in this paper.

Input language description. The `FocusCheck` model checker takes as input programs written in `Cimple`, an expressive subset of C. The basic building blocks of `Cimple` are integer, boolean and array data types, and assignment, conditional (`if`, `while` statements), call and return statements. Due to the absence of pointers and address reference mechanisms, array of size n is treated as n different variables identifi ed by the name of the array appended to the index value of the element; e.g. the third element in an array `arr` is identifi ed by the variable `arr3`. Calls to procedure are treated as call-by-value

Error Conditions	Assumptions	Simple Reachability (sec)	Reachability by Summarizing (sec)
altSep=UPWARD_RA	otherTrackedAlt>ownTrackedAlt upSeparation>downSeparation downSeparation<positiveRAAltThresh	8.76	4.23
altSep=DOWNWARD_RA	ownTrackedAlt>otherTrackedAlt upSeparation≤downSeparation upSeparation≥positiveRAAltThresh	7.44	3.98

Fig. 9. Results of model checking the RA module.

and returns from procedure are explicitly handled by assigning the return value to a pre-specified global variable.

Reachability analyzer. The input to the reachability analyzer is a `Cimple` program and one or more error conditions. Forward reachability searches for all counter-examples in the program, and records the control and data dependencies at each statement present in the search path. Backward reachability analysis of counter-examples identifies the FSSs using dependency information. Operations of FSSs are checked for feasibility using CLP(R), a built-in constraint solver in the XSB tabled logic-programming environment [26]. The primary advantage of using logic programming to implement the reachability analyzer includes the direct implementation of least fixed-point summarization relations `fsum` and `bsum`. The output of the reachability analyzer is a set of focus-statement sequences along with their assumption sets.

Components for post-processing FSSs. Ranker orders FSS-assumption pairs and presents the ordered list to the user (Section 3.1). The user can also provide a subset of input variables that s/he considers as important and Projector identifies the constraints over these variables from the assumptions of each FSS. Finally, the Error Neighborhood Detector (END) employs the technique described in Section 3.2 to localize errors to specific segments in each FSS.

5 Verification of the TCAS Resolution Advisory Module

The *Traffic Alert and Collision Avoidance System* (TCAS) [23] issues commercial airline pilots traffic advisories when one or more aircrafts comes in close proximity (airspace) of the controlled aircraft. We concentrate here on the Resolution Advisory (RA) module of TCAS which is used to identify the safest maneuver for the controlled aircraft in the context of various parameters: relative position of the intruder aircraft, motion trajectory, minimum protected zone for the controlled aircraft, etc. The RA module sets a variable `altsep` to `UPWARD_RA` or `DOWNWARD_RA` depending on whether the preferred safety action of the controlled aircraft is to move to a higher or lower altitude.

We analyzed the RA module (174 lines of C source code[3] [15]) using our `FocusCheck` model checker, using different valuations of `altsep` as error conditions. The objective was to identify various pre-conditions on input variables necessary for specific valuation of `altsep`. The assumptions generated exactly match the pre-conditions and prove

[3] Implementation of RA module only uses the C language constructs that can be handled by `FocusCheck` model checker and as such we are not required to perform any abstraction or transformation of the source code.

the correct behavior of the RA module (Table 9). Another important aspect of the RA module is its control structure (Figure 8(b)): a number of procedures are invoked multiple times from different procedures. Timing results reveal that reachability analysis using summarization outperforms naive reachability analysis based on inlining.

6 Related Work

A number of techniques have recently been proposed to provide users with minimal information required to explain counter-examples resulting from model checking. In [25], the authors introduce the notion of *neighborhood of counter-examples* which can be used to understand the cause of counter-examples. A different approach based on game-theoretic techniques is put forth in [20] where counter-examples are augmented into *free segments* (choices) and *fated segments* (unavoidable). Errors are most likely to be removed by careful selection of free segments.

In [1], errors in programs are localized by identifying the diverging point between a counter-example and a *positive* example; a positive example is a sequence of statements in programs that does not lead to a violation of the property of interest. A similar approach is presented in [14] where errors are localized to program statements absent in all positive examples and present in all counter-examples leading to the same error condition. Based on the idea of detecting the divergence as the cause of the counter-example, [13] has developed a technique that uses a distance matrix and constraint manipulations to pin-point the variable operations that led to the divergence. The technique, however, is applied to one counter-example in the program. In contrast, we present a hierarchy of error explanations by analyzing multiple counter-examples.

7 Conclusion

We have presented a methodology for guiding users to locate program errors and for assisting them in the debugging process. The essence of our technique is to organize and reduce a set of focus statement sequences obtained by variable dependency analysis of all counter-examples present in a program written in `Cimple` programming language.

As future work, we intend to enrich the `Cimple` language with pointer and dynamic memory allocation and concomitantly develop subset-based precise interprocedural flow-sensitive pointer analysis techniques. The `FocusCheck` model checker will be appropriately enhanced to handle these new constructs. We would also like to apply our approach to large code bases to understand various scalability issues of our implementation. The primary concern, in this context, is the exploration of large state space of software systems required to find all possible erroneous paths. One possible solution is to make our technique scalable at the cost of losing completeness. The technique relies on providing a reasonable degree of confidence on model checking results by partial code coverage. Finally, extending our techniques to concurrent programs in order to verify temporal safety and liveness properties is another avenue of future research.

References

1. T. Ball, M. Naik, and S.K. Rajamani. From symptom to cause: Localizing error in counterexample traces. In *Proceedings of POPL*, 2003.
2. T. Ball, A. Podelski, and S.K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *Proceedings of TACAS*, 2002.
3. T. Ball and S.K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of SPIN Workshop*, 2000.
4. T. Ball and S.K. Rajamani. Slam, 2003. http://research.microsoft.com/slam.
5. S. Basu, D. Saha, and S. A. Smolka. Getting to the root of the problem: Focus statements for the analysis of counter-examples. Technical report, SUNYSB, 2004.
6. BLAST. Berkeley lazy abstraction software verification tool, 2003. http://www-cad.eecs.berkeley.edu/∼rupak/blast/.
7. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.
8. T.H. Corman, C.E. Leiserson, , and R.L. Rivest. *Introduction to Algorithm*. MIT Press, 1990.
9. M. B. Dwyer and J. Hatcliff. Slicing software for model construction. In *Partial Evaluation and Semantic-Based Program Manipulation*, 1999.
10. D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of SOSP*, 2003.
11. J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *Proceedings of CAV*, 2001.
12. P. Godefroid. Model checking for programming languages using verisoft. In *Proceedings of POPL*, 1997.
13. A. Groce. Error explanation with distance metrics. In *Proceedings of TACAS*, 2004.
14. A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *Proceedings of SPIN Workshop on Model Checking of Software*, 2003.
15. Aristotle Research Group. Program analysis based software engineering, 2003. http://www.cc.gatech.edu/aristotle/.
16. J. Hatcliff and M. Dwyer. Using the bandera tool set to model-check properties of concurrent Java software. *LNCS*, 2154:39–??, 2001.
17. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of POPL*, 2002.
18. G.J. Holzmann and M.H. Smith. Software model checking: Extracting verification models from source code. In *Proceedings of FORTE*, 1999.
19. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of PLDI*, 1988.
20. H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *Proceedings of TACAS*, 2002.
21. MOPED. A model checker for pushdown systems, 2003. http://www.fmi.uni-stuttgart.de/szs/tools/moped/.
22. J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of the ISP*, 1982.
23. RTCA. Minimum operational performance stardards for traffic alert and collision aviodance system (TCAS) airborne equipment consolidated edition, 1990.
24. A. Rybalchenko. *A Model Checker based on Abstraction Refinement*. PhD thesis, Universitt des Saarlandes, 2002.
25. N. Sharygina and D. Peled. A combined testing and verification approach for software reliability. In *Proceedings of FME*, 2001.
26. The XSB logic programming system, 2003. http://xsb.sourceforge.net.