Fault Detection and Reduced Coverage Using Optimal Search in Structured Domains

Yosi Ben-Asher, Igor Breger, Ilia Gordon
Comp. Sci. Dep. Haifa University, Haifa, Israel

Eitan Farchi
I.B.M. Research Center, Haifa, Israel

November 14, 2005

Coverage of a program P

Coverage requires that the user will find a sequence of inputs $I_1, \ldots, I_k$, until the execution of P on those inputs, $P(I_1), \ldots, P(I_k)$, satisfies a condition. For example, the condition might require that all statements, branches or define-use relations of the program have been executed. The goal of the coverage process is

- to find an input I_j that exposes a fault.
- increases the confidence that P has no fault if the coverage condition is met with no fualt.

Motivation

One of the problems of adequately testing industrial programs by meeting coverage criteria is that the number of coverage tasks is too large.
IDEA: reduced coverage: only one of the complete set of inputs are needed to reveal the bug we shall not find all the inputs.
Intuitively: why cover the whole program lets assume an adversary “hide” the bug at the most “difficult” place in the program. Coverage is reduced to those inputs needed for the search.
Observation: Program must be partitioned to herarcical set of components that can be queried by one input.

Scope

Let C be a coverage criterion defined over a directed graph G_P representation of a program P then reduced coverage is basically an optimal search process over G_P. The graph G_P could be:

- the program's static call graph,
- the program's control flow graph,
- the program's define-use graph, etc.

The coverage criterion C can be: statement coverage, branch coverage, multi-condition coverage, or define-use coverage.

The Overlay partition assumption

- Each node of G_P is associated with a set of statements or **component**, e.g., a function definition in the static call graph.
- If a test of a program component fails and the same test of all its neighbors succeeds the component contains a failure.
- Fixing some coverage criterion, such as branch coverage, a program component might be covered.
- If a component is covered and the bug is not revealed our confidence that the bug is not in that component is increased.

This assumption is used to define the queries of the optimal search algorithm.

General notion of optimal search in structured domains

- At each stage the current stage of the search process is modeled by a set α. such that the 'buggy' element can be any element in α.

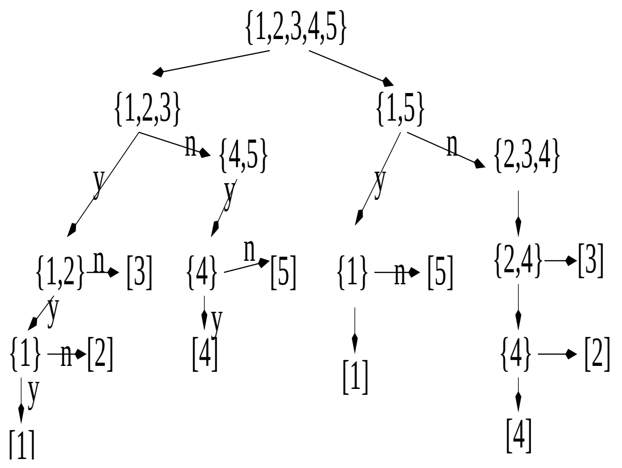
- The possible queries at this stage are some predesignated subsets $\{\beta_1, \ldots, \beta_k\}$, such that $\beta_i \subset \alpha$.

- Each query β_i either directs the search to β_i in the case of a positive answer ($'yes'$) to the query, or directs the search to $\alpha \backslash \beta_i$ in the case of a negative answer ($'no'$).

- This process continues until $|\alpha| = 1$ and a buggy element is located.

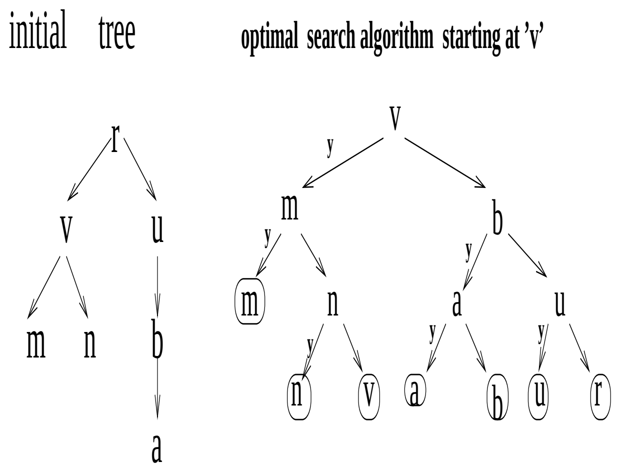
General search algorithms

A search algorithm, denoted Q_D, is a decision tree whose nodes indicate which query should be used at each stage of the search. An optimal search algorithm minimizes the number of queries required to locate any buggy element.



Restricted domains: search in rooted trees

The set of queries possible at each stage of the search is modeled by all the sub-trees of the cuurent tree. If the queried sub-tree does not contain the buggy node, the search continues with the complement tree.



Reduced coverage of P based on optimal search in G_P

We say that P is reduced covered by a set of inputs I if

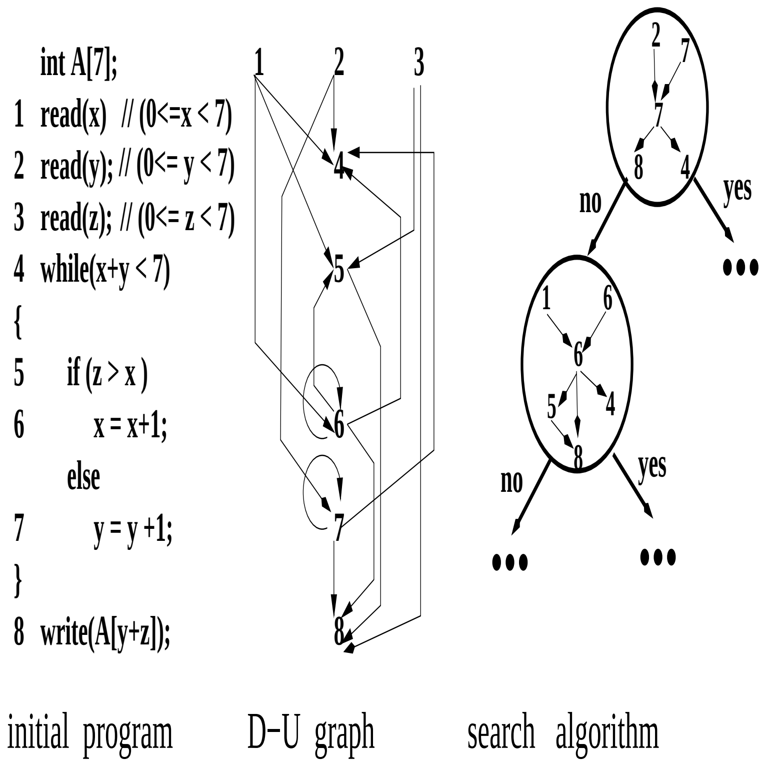
- Let $u_1, \ldots, u_k$ correspond to a **maximal** sequence of queries applied by an optimal search algorithm to G_P (worst case search).
- the coverage criterion C is obtained on $u_1, \ldots, u_k$ by I.

As the number of nodes required to optimally query a directed graph is small compared to the number of nodes in the graph, the resulting number of coverage tasks is also reduced.

Confidence level

If the reduced coverage is met, then it is as if an adversary chose the worst component for us to cover (in terms of the way these components are related to each other in G_P). Next, each of the chosen components are covered according to the coverage criterion C. As a result, and based on the overall partition assumption, our confidence level that the program P does not have a failure increases.

Example Define-Use coverage



```
  int A[7];
1 read(x)  // (0<=x < 7)
2 read(y); // (0<= y < 7)
3 read(z); // (0<= z < 7)
4 while(x+y < 7)
{
5   if (z > x )
6       x = x+1;
    else
7       y = y +1;
}
8 write(A[y+z]);
```

When a bug is detected switch to Algorithmic debugging

Algorithmic debugging is a machine guided search taken by the user to locate a fault in a given run. Based on the user's answare, the algorithm tells the user where to place the next breakpoint and query variables values. This search can be fully automated when appropriate pre-post conditions is available.

The algorithmic debugging phase applies the same optimal search algorithm to a run-time directed graph determined by the program P, such as the dynamic program call graph. In this way the user search for the bug is guided.

Advanteguses of combining reduced coverage and algorithmic debugging

- Using optimal search algorithm optimizes the number of queries compared to the traditional algorithmic debugging.
- Traditional algorithmic debugging starts with a faulty input while the proposed algorithmic debugging can start with a faulty input and an identified faulty component.
- both phases use the same underlying technology; the optimal search algorithm, but on a different directed graphs.

The REDBUG tool

REDBUG is currently designed to work with call coverage and contains the following components:

1. An instrumentation module that can generate the static call graph C_P.

2. C_P is converted to a tree by selecting a spanning tree of C_P.

3. For a given input I of P, this module can also produce the dynamic call graph $C_{P(I)}$.

4. For call coverage, $C_{P(I)}$ is a rooted tree.

5. The search module computes the optimal search algorithm A_{C_P}.

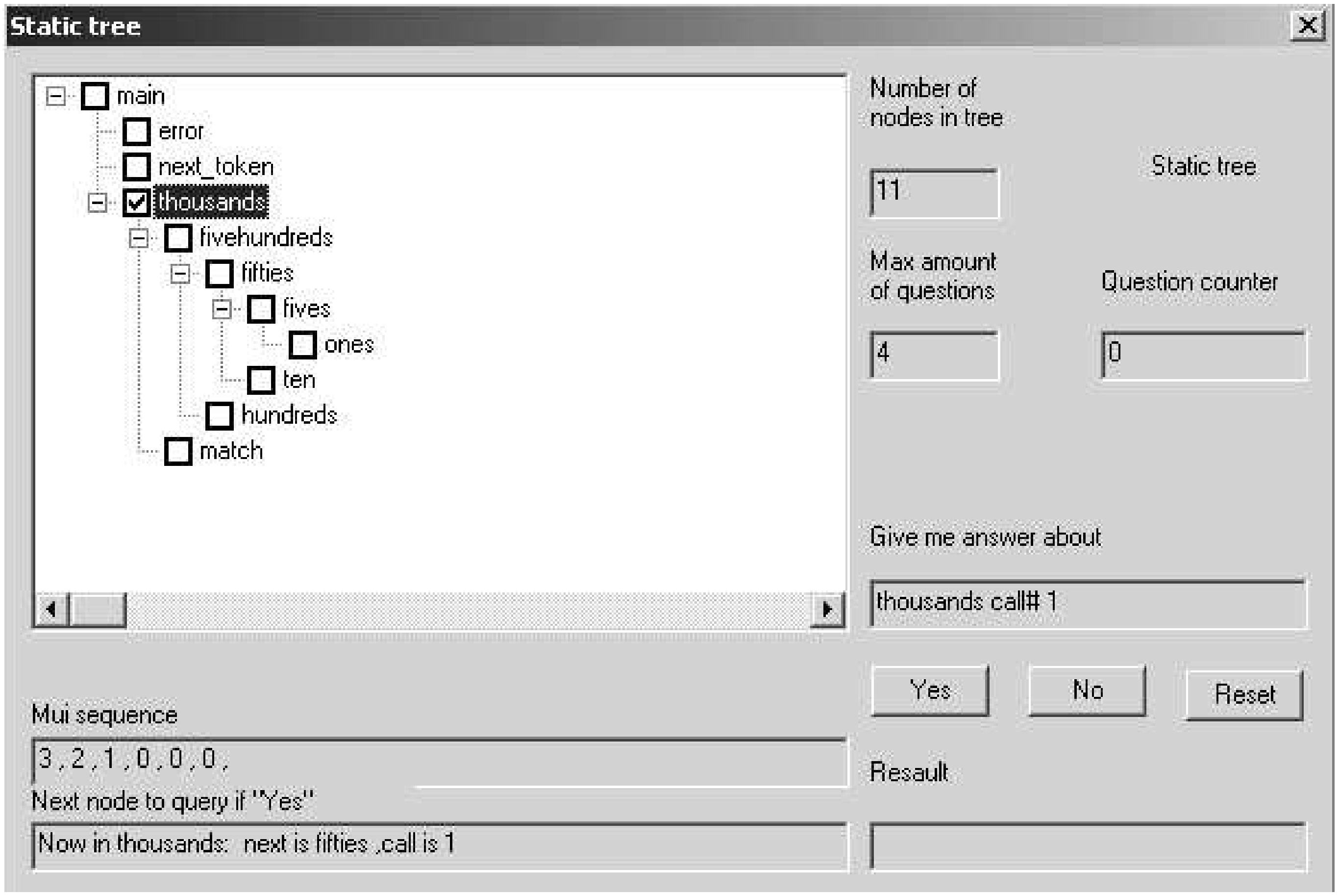
6. An ineractive procedure to applay A_{C_P}, requesting input to cover the next queried component C_P *or* $C_{P(I)}$.

7. Algorithmic debugging is activated when a fualt occures with input I.

REDBUG's algorithmic debuging

- The debugger is used to determine if a given call to a function $f(...)$ is buggy or not.
- We use the debugger breakpoint mechanism to locate a specific call to a function in $C_{P(I)}$.
- we use the debugger to check the values of $f(...)$'s variables and see if their value is as expected.
- The debugger is guided by the optimal search algorithm of $A_{C_{P(I)}}$ computed on the fly when the fualt is detected.

Screenshot 1: Search tree



Number of nodes in tree: 11

Static tree

Max amount of questions: 4

Question counter: 0

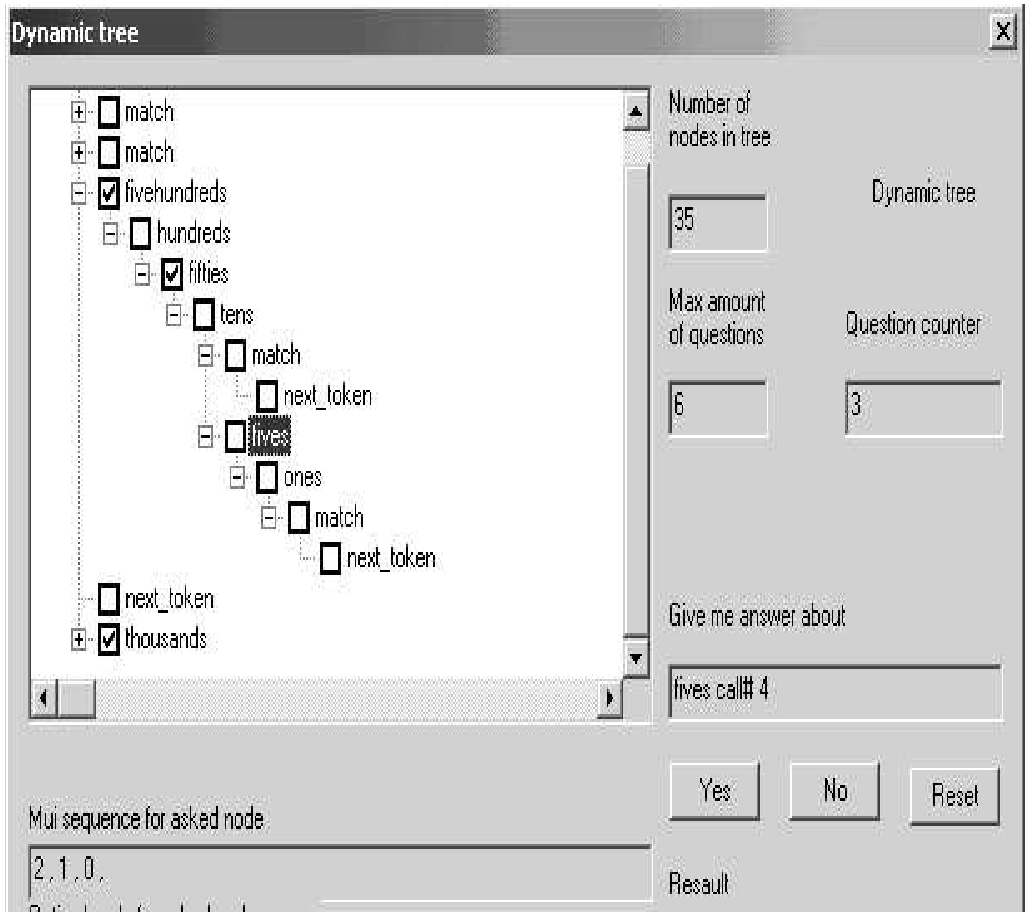
Give me answer about: thousands call# 1

Mui sequence: 3 , 2 , 1 , 0 , 0 , 0 ,

Next node to query if "Yes": Now in thousands: next is fifties ,call is 1

Resault:

Screenshot 2: Next query



- ☐ match
- ☐ match
- ☑ fivehundreds
  - ☐ hundreds
    - ☑ fifties
      - ☐ tens
        - ☐ match
          - ☐ next_token
        - ☐ fives
          - ☐ ones
            - ☐ match
              - ☐ next_token
- ☐ next_token
- ☑ thousands

Number of nodes in tree: 35

Dynamic tree

Max amount of questions: 6

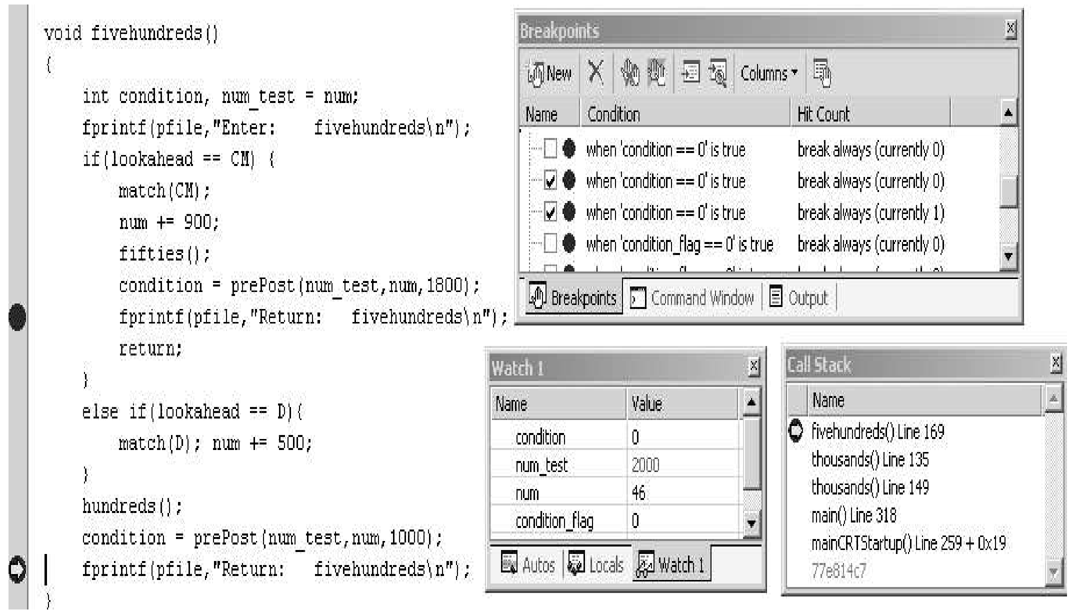
Question counter: 3

Give me answer about: fives call# 4

Mui sequence for asked node: 2,1,0,

Resault

Screenshot 3: Using the debugger to evaluate a query



```
void fivehundreds()
{
    int condition, num_test = num;
    fprintf(pfile,"Enter:   fivehundreds\n");
    if(lookahead == CM) {
        match(CM);
        num += 900;
        fifties();
        condition = prePost(num_test,num,1800);
        fprintf(pfile,"Return:   fivehundreds\n");
        return;
    }
    else if(lookahead == D){
        match(D); num += 500;
    }
    hundreds();
    condition = prePost(num_test,num,1000);
    fprintf(pfile,"Return:   fivehundreds\n");
}
```

Name	Condition	Hit Count
☐ ●	when 'condition == 0' is true	break always (currently 0)
☑ ●	when 'condition == 0' is true	break always (currently 0)
☑ ●	when 'condition == 0' is true	break always (currently 1)
☐ ●	when 'condition_flag == 0' is true	break always (currently 0)

Name	Value
condition	0
num_test	2000
num	46
condition_flag	0

Name
fivehundreds() Line 169
thousands() Line 135
thousands() Line 149
main() Line 318
mainCRTStartup() Line 259 + 0x19
77e814c7