Optimal Algorithmic Debugging and Reduced Coverage Using Search in Structured Domains

Yosi Ben-Asher
Comp. Sci. Dep. Haifa University, Haifa, Israel

Igor Breger
Comp. Sci. Dep. Haifa University, Haifa, Israel

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

Ilia Gordon
Comp. Sci. Dep. Haifa University, Haifa, Israel

October 21, 2005

Abstract

Traditional code based coverage criteria for industrial programs are rarely met in practice due to the large size of the coverage list. In addition, debugging industrial programs is hard due to the large search space. A new tool, REDBUG, is introduced. REDBUG is based on an optimal search in structured domain technology. REDBUG supports a reduced coverage criterion rendering the coverage of industrial programs practical. In addition, by using an optimal search algorithm, REDBUG reduces the number of steps required to locate a bug. REDBUG also combines testing and debugging into one process.

1 Introduction

One of the problems of adequately testing industrial programs by meeting coverage criteria is that the number of coverage tasks is too large. For example, covering all the define-use relations of a given industrial program might prove impractical, and indeed such coverage criteria are rarely met in practice.

Given a program based coverage criterion [11] defined on some directed graph determined by the program under test P, we propose a corresponding, intuitively appealing, reduced coverage criterion. The directed graph could be the program's static call graph, the program's control flow graph, the program's define-use graph, etc. The coverage criterion can be any one of the standard criteria defined on such graphs such as: statement coverage, branch coverage, multi-condition coverage, or define-use coverage ([4], [5], [10], [11]).

Reduced coverage requires meeting a much smaller coverage task list and is thus more practical than the standard coverage criteria. Further research is required to determine the confidence level achieved by this new set of reduced coverage criteria.

Fixing some code based coverage C, a process for obtaining a C reduced coverage criterion and a process, called algorithmic debugging, for assisting the programmer in debugging the implementation ([6], [9], [8], [7], [3]) are described. Then a method for combining both processes in one tool, called Reduced Coverage and Algorithmic Debugging, REDBUG, is described. Both processes leverage an optimal search algorithm defined on the relevant program directed graph ([2], [1]).

In large industrial programs debugging is a tedious and time consuming task due to the large size of the search space. We address this problem by applying an optimal search algorithm to a run-time directed graph representing the program P. By doing this, we improve REDBUG algorithmic debugging compared to traditional algorithmic debugging and reduce the number of queries required to locate the fault.

This paper is organized as follows. First, search in structured domains is introduced. Next, reduced coverage and optimal algorithmic debugging is defined. Finally, a detailed example is given.

2 Search in Structured Domains

For the purpose of defining the concept of reduced coverage criteria and the optimal algorithmic debugging process, we first explain the general notion of an optimal search algorithm in structured domains. Searching in structured domains is a staged process whose goal is to locate a 'buggy' element. The current stage of the search process is modeled by a set α. At the current stage, the 'buggy' element can be any element in α. The possible queries at this stage α are modeled by a set of sets $\{\beta_1, \ldots, \beta_k\}$, 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. 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 types of structured search domains, such as trees or directed graphs, have been studied in [2]. In [2], it was shown that an optimal search algorithm for trees, Q_D, can be computed in $O(|D|^4)$ where D are the tree's nodes. In the case of a tree, the current stage of the search process is modeled by a sub-tree. The set of queries possible at each stage of the search is modeled by all the sub-trees of this tree, and an available sub-tree is queried at each stage of the search. If the queried sub-tree does not contain the buggy node, the search continues with the complement tree. An example of an optimal search algorithm for a tree is given in figure 1. The arrows point to the next query. This search algorithm takes three queries in the worst case. Any other search which does not start at node $'v'$ takes at least four queries in the worst case[1].

[1]Intutivaly, $'v'$ separates the tree in the "middle". Consider choosing $'u'$ instead of $'v'$ as $'u'$ also, intuitively, separates the tree in the "middle". An adversary would chose to answer $'no'$. Next, you best chose $'v'$. This time an adversary would answer $'yes'$ resulting in two more queries, e.g., $'m'$ answered by $'no'$ and then $'n'$ answered by $'no'$. Overall, four queries were

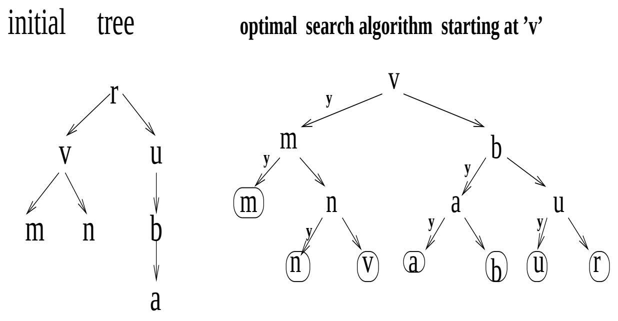


Figure 1: *Searching in a tree.*

3 Reduced Coverage and Optimal Algorithmic Debugging

Coverage and algorithmic debugging are next defined.

Coverage - Coverage of a program P, 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 in P or
- to give evidence that increases the confidence that P has no fault if the coverage condition is met and no failing run, $P(I_j)$, occurred.

Algorithmic debugging - Algorithmic debugging is a machine guided search taken by the user to locate a fault in the run $P(I_j)$ once the run $P(I_j)$ fails. The algorithm tells the user where to place the first breakpoint and query variable's value. Then, based on some user feedback, the algorithm determines the program location of the next breakpoint to be placed by the user, and so forth. This search can be fully automated when a database of appropriate pre-post conditions is available.

We are now ready to define reduced coverage. We are given a program P, a directed graph, G_P, defined by P and a code based coverage criterion C. We are further given that each component, or node, u_i of the graph G_P corresponds to a subset of P's statements. We use the notion of an optimal search of G_P to define a reduced coverage criterion. We say that P is reduced covered by a set of inputs I if

- $u_1, \ldots, u_k$ correspond to an optimal sequence of queries applied by the optimal search algorithm when G_P is treated as a structured search domain and the "buggy" element is not found.
- the coverage criterion C is obtained on $u_1, \ldots, u_k$ by I.

used which is worst than the number of queries that result in the worst case when choosing $'v'$.

As the number of nodes required to optimally query a directed graph is small compared to the number of nodes in the graph, the number of coverage tasks to cover, such as code branches to cover, is greatly reduced rendering the task of obtaining a reduced coverage criterion practical.

If the reduced coverage is met, then it is as if an adversary chose the worst components 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 our confidence level that the program P does not have a fault increases.

Applying our method, the optimal search algorithm is given a directed graph G_P determined by the program and output a list of components to be covered $u_1, \ldots, u_k$. The programmer attempts to cover each component, u_i, according to the coverage criterion C. If a failure occurs the algorithm debugging stage is invoked guiding the programmer in the debugging of the failure. 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. As an optimal search is used to guide the algorithmic debugging stage, the number of breakpoints to set an inspect by the user is greatly reduced rendering the debugging task of large industrial components simpler.

In REDBUG, reduced coverage and the new algorithmic debugging process were implemented for the static and dynamic call graph of the program. The rest of the paper concentrates on the details of the implementation.

4 Detailed example

We demonstrate how REDBUG is used to obtain reduced coverage and debug program failures. 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 of a given program P. The resulting static graph C_P is converted to a tree by selecting a spanning tree of C_P. For a given input I of P, this module can also produce the dynamic call graph $C_{P(I)}$. Note that for call coverage, $C_{P(I)}$ is in fact a tree, as each call to a function has a single caller. Thus, we refer to C_P as the static tree of P and to $C_{P(I)}$ as the dynamic tree of $P(I)$.

2. The search module, which computes the optimal search algorithm A_{C_P} for the static tree of P and $A_{C_{P(I)}}$ for the dynamic tree of P. This module is interactive, and based on the answer for the last query (fed to it by the user) it prompt the next node to query in C_P *or* $C_{P(I)}$.

3. Finally, 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 $C_{P(I)}$. Once a suitable breakpoint is reached, we use the debugger to check the values of $f(...)$'s variables and see if their value is as expected. In the following example, we inserted a check for the validity of a pre-post condition in every function that is queried. We prompt $'yes'$ if the pre-post condition of a given call is $'false'$, meaning that failure is in that call and $'no'$ otherwise.

We have chosen a simple program[2] that computes the value of roman numbers. For the input "MIX" the program should compute 1009, for "MXI" the program should compute 1011, etc. Many combinations are not allowed, e.g., $'MXM', 'LL', 'IIIVII'$ for which an error message should be printed.

The program implements the grammar of roman numbers using the following routines:

- thousands() - converts each 'M' to +1000
- fivehundreds() - converts 'CM' to +900 and each 'D' to +500
- hundreds() - converts 'CD' to +400 and each 'C' to +100
- fifties() - converts 'XC' to +90 and each 'L' to +50
- tens() - converts 'XL' to +40 and each 'X' to +10
- fives() - converts 'V' to +5
- ones() - converts 'IX' to +9, 'IV' to +4 and each 'I' to +1
- match(), next_token() and error() - process the input and report errors

A simple pre-post condition function $prePost(int\ before, int\ after, int\ max\_inc)$ is used to check if a given call to a function contains a bug. This is done to leverage the coverage process. When during the coverage process a test fails we immediately know which function is failing and the debugging phase is simplified. Obviously, the proposed notion of query depends on the ability of the programmer to insert such tests at suitable places in the code. Due to the way functions to be covered are chosen in the reduced coverage process , the list of functions to cover is usually small. As a result, the task of inserting pre-post conditions for the list of functions to cover becomes easier.

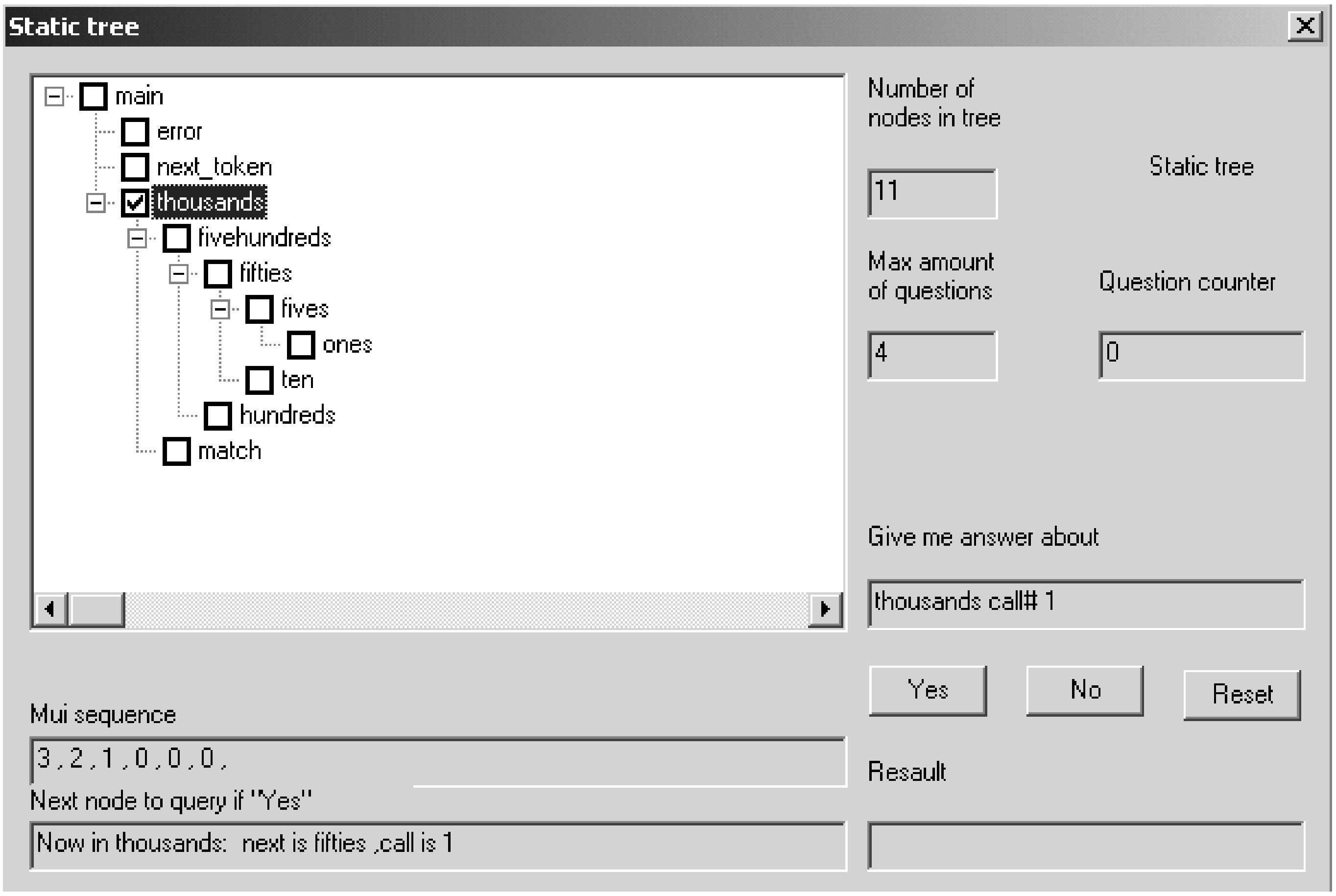
First instrumentation is applied to generate the static call tree of the program. The search module is then applied on the resulting tree. Figure 2 is a screen shot of the the search module showing: the static tree, and the first query in an optimal search of the call tree. Thus, the first function to cover is $thousands()$. We can see that we need to cover only four functions out of the total of eleven functions.

Next we select the inputs to cover $thousands()$, these inputs contain three roman numbers: $XXVII$ for which $thousands()$ outputs the correct value of 27, $MMXLVI$ for which $thousands()$ outputs the wrong value of 46 (instead of 2046) and $MMXXXIV$ for which $thousands()$ outputs the correct value 2034.

In order to locate and correct the bug, we move to the algorithmic debugging phase, generate the dynamic tree and apply the search tool on the dynamic tree (starting from the first call to $thousands()$). The dynamic tree contains 35 nodes and as is depicted in figure 3 can be searched in at most six queries. Figure 3 describes the situation after three queries on the dynamic tree. At this stage, we pressed 'yes' to the first query on $thousands()$, 'no' to the second query on $thousands()$ (marked at the bottom of the window in figure 3), 'yes' to $fivehundreds()$, and 'yes' to $fifties()$. We are now advised to query $fives()$, call number 4 and see if it is erroneous or not (see figure 3).

We evaluate each query to a specific call using the debugger. First we insert a call to $prePost()$ before the function returns and in-

[2]The program was written by Terry R. McConnell.



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:

Figure 2: *Search tree for Roman.c .*

sert a breakpoint after the calls to $prePost()$. By examining the result of the pre-post conditions and the value of other variables we are able to decide if the current call executed correctly or not. For example, in figure 4 we can check the return value of $prePost()$ and other variables using the $Watch$ window. In figure 4 we have two active breakpoints, and the debugger stops with a false *condition* from $prePost(num\_test, num, 1000)$, so that this call is buggy. The debugger is also used to reach a specific call , e.g., $fives()$, call number four, of the dynamic tree. This is done using a global counter in the code which is incremented every time $fives()$ is called. The breakpoint is set to stop only if the value of this counter is the desired one (four in the case of $fives()$, call number four).

We use the debugger to evaluate call number four, of $fives()$ and find out that the pre-post condition is true. Consequently we prompt $'no'$ to the search tool. As can be seen from

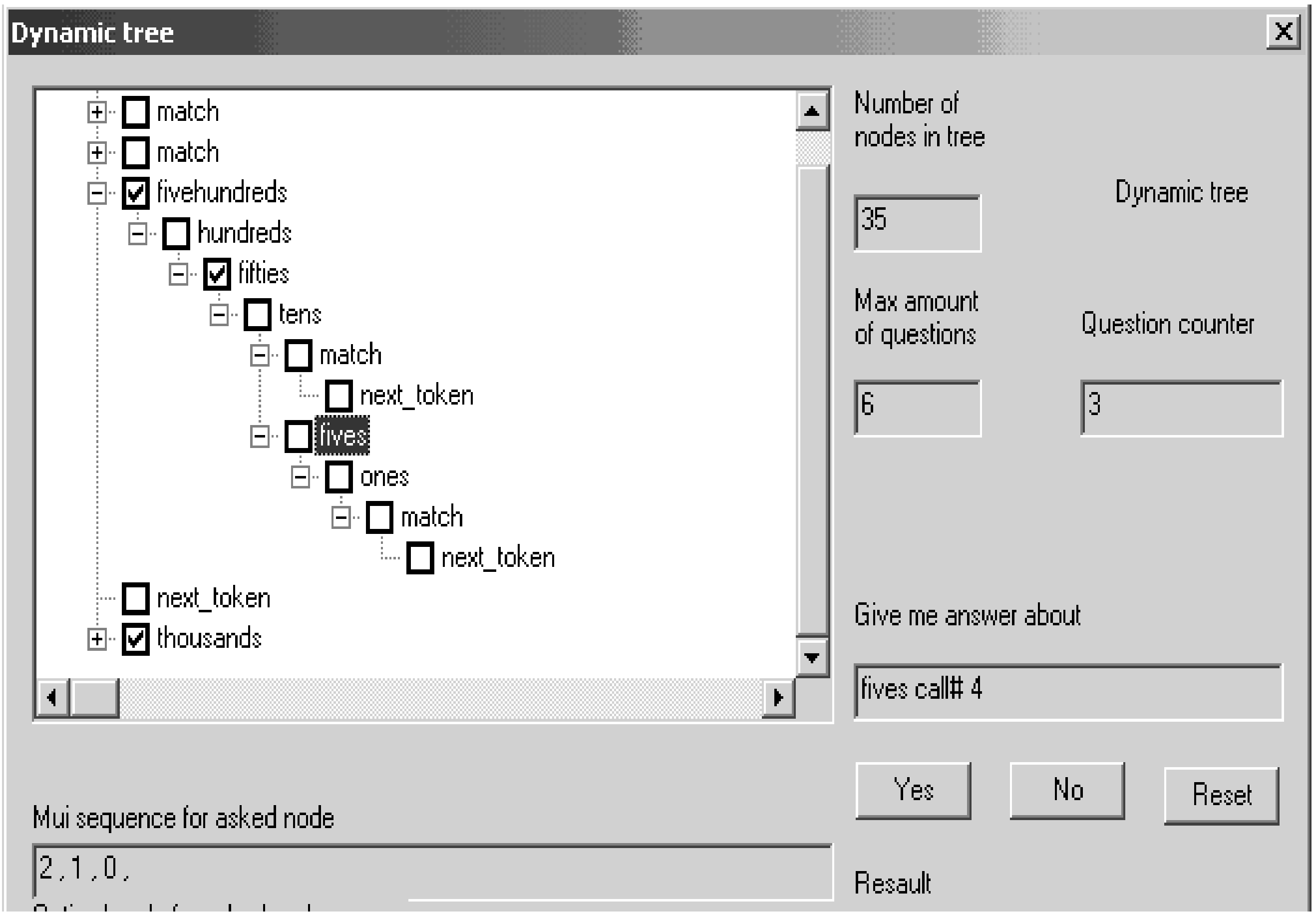
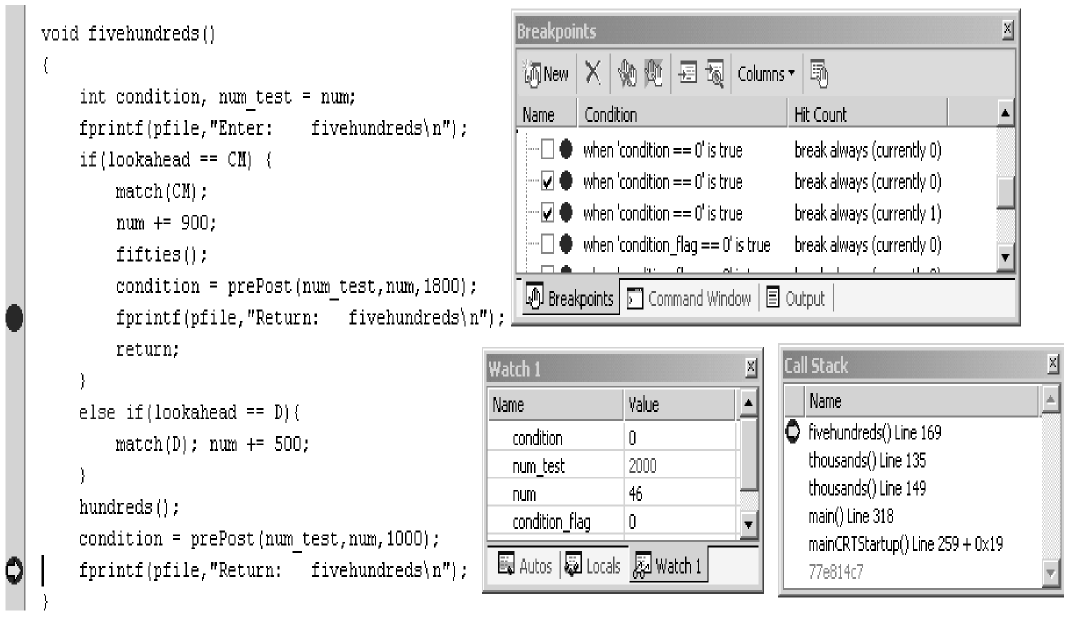


Figure 3: *Next query is* $fives()$ *call no.* 4.

the tree in figure 3, the next query will be on $match()$ for which we also prompt $'no'$ (after checking the program state using the debugger). Thus, the search process ends by locating the bug in $tens()$. We use the debugger, one last time to examine the pre-post condition and try to locate the bug. Figure 5 depicts this stage, the call to $prePost(num\_test, num, 80)$ return false, as initially the value accumulated so far is $num\_test = 2000$, yet the value returned by $tens()$ which should have been between num_test and $num\_test + 80$ has been reduced to $num = 40$. As we know that the call to $fives()$ is correct, we find the bug in statement $num = 40$ that should have been $num += 40$.

After fixing that bug, we continue in the reduced coverage process and as no further bugs are found the process ends. This process obtained reduced call coverage using three inputs and used six tests to locate the failure.



```
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");
}
```

Breakpoints

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)

Watch 1

Name	Value
condition	0
num_test	2000
num	46
condition_flag	0

Call Stack

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

Figure 4: *Using the debugger to evaluate a query.*

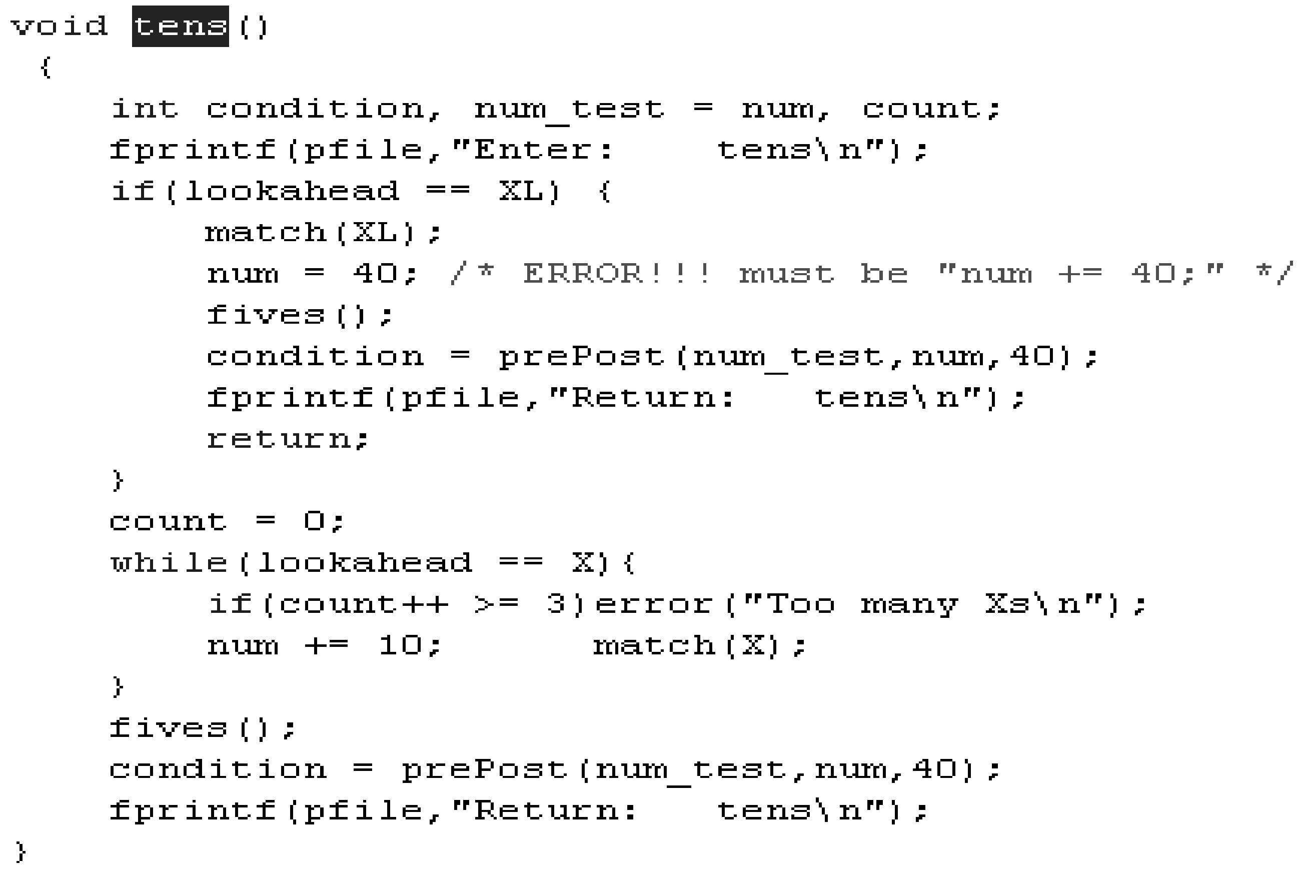
5 Conclusion

A tool called REDBUG that combines software testing and algorithmic debugging is introduced. The tools uses the same underlying technology of optimal search in structured domains to support the testing and debugging phases.

Traditional code based coverage criteria have large coverage task list rendering them impractical. The notion of an optimal search in structure domain is used to define a new set of reduced coverage criteria corresponding to the traditional coverage criteria rendering the coverage of industrial programs practical. If a reduced coverage criteria is met, then it is as if an adversary chose the most complex components for the user to cover. More research is required to determine the confidence level these new reduced coverage criteria provide.

In large industrial components the problem of locating a failure requires many hours of setting breakpoints and observing of program behavior. By using an optimal search algorithm, REDBUG is able to reduce the number of breakpoints a programmer should set in-order to find the bug. REDBUG will be especially useful in setting were there are the call stack is big or when there are a lot of communication layers resulting in large search domains.

Further research will implement REDBUG on other program defined graphs. It would be interesting to see how much saving is obtained by the new algorithmic debugging method on a set of real life industrial examples. Finally, the issue of the confidence level that the new set of reduced coverage criteria introduce should be addressed.



```
void tens()
 {
     int condition, num_test = num, count;
     fprintf(pfile,"Enter:     tens\n");
     if(lookahead == XL) {
         match(XL);
         num = 40; /* ERROR!!! must be "num += 40;" */
         fives();
         condition = prePost(num_test,num,40);
         fprintf(pfile,"Return:    tens\n");
         return;
     }
     count = 0;
     while(lookahead == X){
         if(count++ >= 3)error("Too many Xs\n");
         num += 10;        match(X);
     }
     fives();
     condition = prePost(num_test,num,40);
     fprintf(pfile,"Return:    tens\n");
}
```

Figure 5: *Using the debugger to locate the bug in tens*().

References

[1] Y. Ben-Asher and E. Farchi. Compact representations of search in complex domains. *International Game Theory Review*, 7(2):171–188, 1997.

[2] Y. Ben-Asher, E. Farchi, and I. Newman. Optimal search in trees. In *8'th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA97), New Orleans*, 1997.

[3] Jong Deok Choi and Andreas Zeller. Isolating failure inducing thread schedules. *International Symposium on Software Testing and Analysis*, 2002.

[4] L. A. Clarke. comparison of data-flow path selection criteria. *IEEE Transaction on Software Engineering*, 1985.

[5] L. A. Clarke. An investigation of data flow path selection criteria. *Work Shop*

On Software Testing, Banff, Canada, 1986.

[6] Peter Fritzson et. al. Generalized algorithmic debugging and testing. *ACM Letters on Programming Languages and testing*, 1:303–322, 1992.

[7] Fritzson Peter Kamkar Mariam and Shahmehri Nahid. Interprocedural dynamic slicing applied to interprocedural data flow testing. In *Proceedings of the Conference on Software Maintenance*, 1993.

[8] Nahid Shahmehri Mariam Kamkar and Peter Fritzson. Interprocedural dynamic slicing and its application to generalized algorithm debugging. In *Proceedings of the International Conference on Programming Language*, 1992.

[9] Mikhail Auguston Peter Fritzson and Nahid Shahmehri. Using assertions in declarative and operational models for automated debugging. *Journal of Systems and Software*, 25(3):223–232, June 1994.

[10] Elaine J. Weyuker. Axiomatizing software test data adequacy. *IEEE Transaction on Software Engineering*, SE-12(12), December 1986.

[11] Elaine J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM*, 31(6), June 1988.