Automated Program Debugging for Multiple Bugs Based on Semantic Analysis

Aishan Liu(✉), Li Li, and Jie Luo

Department of Computer Science, Beihang University, Beijing 100191, China
{liuaishan,lili,luojie}@nlsde.buaa.edu.cn

Abstract. Fault locating is a time-consuming process. In a previous paper, Liu proposed an algorithm named bounded debugging via multiple predicate switching (BMPS)[1], which try to find a successful execution trace by switching outcomes of multiple predicates. Substantively, BMPS focuses on the program faults which are caused by control flow. However, this kind of faults represent only a small fraction. In this paper, we present an algorithm combining BMPS with a semantic based debugging method, which is aimed at locating more than control flow related faults. The semantic based debugging algorithm generates a sequence of equations from the execution trace of a failed test case, and give a minimum faulty program segment according to solutions of the equations. Our algorithm can locate multiple faults in the program one by one through an iterative and interactive process. Moreover, an optimization based on use-define chain is applied to BMPS for improving efficiency of the algorithm, as well as some other methods. To evaluate out approach, we conduct experiment on Siemens suite. The result indicates that our method has significant improvement on both accuracy and efficiency.

Keywords: Automated debugging · Predicate switch · Semantic analysis

1 Introduction

Traditional program debugging is an arduous and manual process that requires a large number of time, effort, and a well comprehension of the code. Generally, traditional debugging consists of two steps: (1) program turns out an unexpected result or behavior; (2) developer re-executes the failed test case step by step and inspects the program state to find out the cause of the problem. Such process can be apparently time consuming, tedious and sometimes error-prone.

Researchers were more interested in utilizing computer power for program debugging since the processing power of machines has drastically increased. Therefore automated debugging techniques are being explored by researchers. Automated debugging basically consists of program slice based model, program state based model, statistical model and program spectrum based model [2,3].

In a previous paper [4], Zhang proposed an algorithm which switches the outcome of an instance of a single predicate and then inspect the so called

S. Liu and Z. Duan (Eds.): SOFL+MSVL 2015, LNCS 9559, pp. 86–100, 2016.
DOI: 10.1007/978-3-319-31220-0_7

switched predicate to find out the cause of the fault. After experiment evaluation, the author found their approach to be effective and practical. However, switching a single predicate could not be sufficient and useful in some situation. In 2010, Liu proposed the bounded debugging via multiple predicate witching algorithm (BMPS) which is to switch the outcome of instances of multiple predicates to locate bugs. However, both the algorithms are basically aimed at control flow related program bugs, and cost much time especially when the program contains amounts of predicates (e.g. conditional statements).

In this paper, we propose a method combining BMPS with Intra-function debugging algorithm (Sect. 2.2), which is aimed at locating multiple bugs including not only control flow related faults. We present an iterative and interactive approach for programmer to locate program bugs, which would provide developer with a minimum program segment containing faulty code. Some program would contain a lot of predicates, and many of them could be unrelated to the results or the variables with unexpected value. If we treat every predicate to be switchable, it would be really time consuming and redundant. So, we optimize the algorithm by applying use-define chain to reduce the number of predicates to be switched. Since our algorithm is interactive, programmer could mark some statements as errorless such that workload of predicate switching will be reduced.

2 Technical Background

2.1 BMPS

Bounded debugging via multiple predicate switching algorithm (BMPS) locates faults through switching the outcomes of instances of multiple predicates to get a successful execution where each loop is executed for a bounded number of times. The most important concept is critical predicate set. It is a set of predicates which could be switched to get the right execution or result of the program. Obviously, BMPS concentrated on the control flow related bugs in the program. By switching the outcomes of predicates, which could be the branch statements, the execution trace would go to another path leading to a right output under specified test case.

Figure 1 below depicts a small program example. In this program, inputs variables are a and b. However, the conditional statement in line 4 is faulty. Test

```
1.  int i=5,x;
2.  int a,b;
3.  read(a,b)
4.  if(i<a+b) //correct version: i<=a+b
5.    x=i+1;
6.  else
7.    x=i;
```

Fig. 1. Example of a Program.

case of $< a = 2, b = 3, x = 6 >$ is failed. As for this test case, the expected output is x=6, yet x is 5 after execution. If we switched the outcome of the predicate $i < a + b$ to $!(i < a + b)$, the execution trace could change from (1,2,3,7) to (1,2,3,5) which would lead to the right result. Thus, predicate $i < a + b$ is a critical predicate by which we could switch to get the right result.

For iterative programs, critical predicate set should be uncomputable. So the author considered depth k critical predicate set which is deduced from an execution path where each loop is executed at most k times.

For each predicate p in the program, we introduce a Boolean variable sw_p, and replace p with $\neg sw_p \wedge p \vee sw_p \wedge \neg p$. Given a program, we define the formula as below:

$$\Phi = IN \wedge S_1 \wedge ... \wedge (\neg sw_p \wedge p \vee sw_p \wedge \neg p) \wedge ... \wedge S_n \wedge OUT$$

IN is the input of the test case, S_i denotes the statements in the program and OUT means the assertion of the test case (usually the output or the result of the test case). After that, we convert Φ to conjunctive normal form (CNF) and feed it a SAT solver (e.g., z3, minisat, etc.) to solve the satisfiability problem. If the CNF is satisfiable, and the value of sw_p is true, which means the value of p is false, the execution trace goes to another path leading to the right result. Thus, the assertion holds and p could be a critical predicate to reveal the bugs.

Obviously, it is more convenient to switch the outcomes of some predicates if whole program is transferred to CNF [5]. Then, we resort to a SAT solver (e.g., z3, minisat, etc.) to solve the satisfiability problem.

2.2 Intra-function Debugging Algorithm

First of all, we give some brief definition to help depict our algorithm.

Definition 1 *(State). A state σ is defined as a k-tuple.*

$$\sigma : (x_1 \rightarrow n_1, ..., x_k \rightarrow n_k)$$

$x_i \rightarrow n_i$ means the value of x_i is n_i, and denoted as $\sigma(x_i) = n_i$

Definition 2 *(Environment). Assume S is the head of a conditional statement, and σ is the state when executing S. $\langle S, \sigma \rangle$ is called an Environment.*

Definition 3 *(Structure-S). Assume σ is a state, S is the statement executed under the state σ, and ϵ is the stack of Environment. Thus, the triple below is a Structure-S:*

$$\langle S, \sigma, \epsilon \rangle$$

Definition 4 *(Trace Stack). A stack is called a Trace Stack iff every element in the stack has the type of Structure-S.*

Definition 5 *(Structure-T). Assume $\langle S, \sigma, \epsilon \rangle$ is a Structure-S, π is the corresponding trace stack. A structure-T is denoted as below:*

$$\pi | \langle S, \sigma, \epsilon \rangle$$

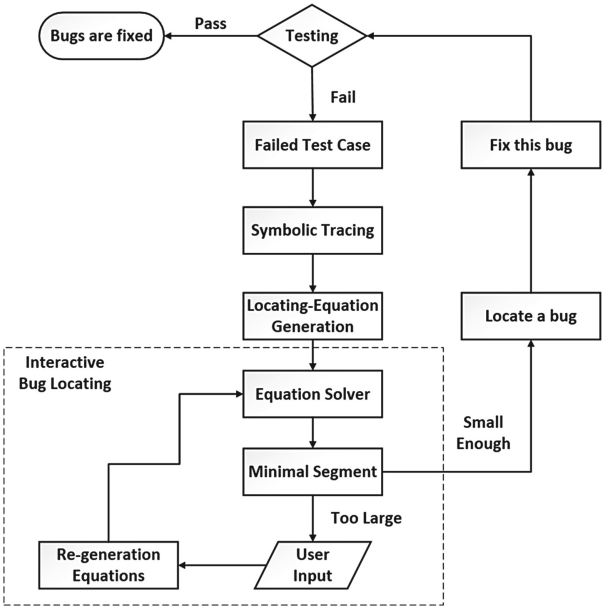

Fig. 2. Flow Diagram of Intra-function Debugging Algorithm.

Intra-function debugging algorithm uses semantic analysis to give developer a minimal faulty program segment. When a test case failed, the algorithm computes a minimal inconsistent program segment according to the input and output of the test case. Programmers are able to mark some statements to be errorless iteratively to reduce the scale of faulty program segment.

Figure 2 is the flow diagram of Intra-function debugging algorithm, which simply shows the flow and main steps of the method.

The algorithm includes four main steps:

(1) *Symbolic Execution*. Execute and trace the program by symbolic representation according to the input of the failed test case. Record the execution trace and program states.

The execution of program could be treated as many steps of compute. And every step is similar to a transition from a Structure-T to another.

-Assignment

$$\pi|\langle l : x = e, \sigma, \epsilon\rangle \rightarrow push(\langle l : x = e, \sigma, \epsilon\rangle)|\langle skip, \sigma[[e]_\sigma/x], \epsilon\rangle$$

$[e]_\sigma$ is the value of expression e under state σ.

-Sequence of Statemtents

$$\frac{\pi|\langle S_1, \sigma, \epsilon\rangle \rightarrow \pi'|\langle S_1', \sigma', \epsilon'\rangle}{\pi|\langle S_1; S_2, \sigma, \epsilon\rangle \rightarrow \pi'|\langle S_1'; S_2, \sigma', \epsilon'\rangle}$$

$$\pi|\langle skip; S, \sigma, \epsilon\rangle \rightarrow \pi|\langle S, \sigma, \epsilon\rangle$$

-Conditionals
If $[b]_\sigma = true$

$$\pi|\langle l : if(b)\{S_1;\}else\{S_2\}, \sigma, \epsilon\rangle \rightarrow$$
$$push(\langle l : if(b), \sigma, \epsilon\rangle, \pi)|\langle\{S_1;\}, \sigma, push(\langle l : if(b), \sigma\rangle, \epsilon)\rangle$$

If $[b]_\sigma = false$

$$\pi|\langle l : if(b)\{S_1;\}else\{S_2\}, \sigma, \epsilon\rangle \rightarrow$$
$$push(\langle l : if(b), \sigma, \epsilon\rangle, \pi)|\langle\{S_2;\}, \sigma, push(\langle l : if(b), \sigma\rangle, \epsilon)\rangle$$

-Loops
If $[b]_\sigma = true$

$$\pi|\langle l : while(b)\{S;\}, \sigma, \epsilon\rangle \rightarrow push(\langle l : while(b), \sigma, \epsilon\rangle, \pi)$$
$$|\langle\{S;\}l : while(b)S;, \sigma, push()\langle l : while(b), \sigma\rangle, \epsilon)\rangle$$

If $[b]_\sigma = false$

$$\pi|\langle l : while(b)\{S;\}, \sigma, \epsilon\rangle \rightarrow$$
$$push(\langle l : while(b)\{S;\}, \sigma, \epsilon\rangle, \pi)|\langle skip, \sigma, \epsilon\rangle$$

(2) *Locating-Equation Generation.* Generate a set of locating equations for every statement backwards from the beginning of breakpoint - a statement in which the value of specified variable is not expected.

In the second step of the algorithm, we generate locating-equation for every Structure-S in Trace Stack π. Thus we can specify the transition of Structure-S by solving equation set. Besides, the locating-equation sets are stored in a list, in which every element has a form of $(eq, \langle S, \sigma, \epsilon\rangle)$. Among them, eq denotes an equation set while $\langle S, \sigma, \epsilon\rangle$ is a Structure-S.

-Initial State

$$\pi|\rho$$

-Assignments

$$\frac{top(\pi) = \{l : x = e, \sigma, \epsilon\} last(\rho) = (eq, \{S, \sigma', \epsilon'\}) x \in FV(eq)}{\pi|\rho \rightarrow pop(\pi)|append((eq[e/x], \{l : x = e, \sigma, \epsilon\}), \rho)}$$

$$\frac{top(\pi) = \{l : x = e, \sigma, \epsilon\} last(\rho) = (eq, \{S, \sigma', \epsilon'\}) x \notin FV(eq)}{\pi|\rho \rightarrow pop(\pi)|\rho}$$

Given a Structure-S containing statement $l : x = e$ at the top of trace stack. If x appears in equation set eq, all the variable x in eq corresponding current Structure-S can be replaced with e. On the contrary, skip it.

-Conditionals

$$\frac{top(\pi) = \{l : if(b), \sigma, \epsilon\} last(\rho) = (eq, \{S, \sigma', \epsilon'\})}{\pi|\rho \rightarrow pop(\pi)|append((eq \cup \{b = [b]_\sigma\}, \{l : if(b), \sigma, \epsilon\}), \rho)}$$

Given a Structure-S containing head part $if(b)$ of the conditional statement, equation $b = [b]_\sigma$ is added to the equation set corresponding with the current Structure-S.

-Loops

$$\frac{top(\pi) = \{l : while(b), \sigma, \epsilon\} last(\rho) = (eq, \{S, \sigma', \epsilon'\})}{\pi|\rho \rightarrow pop(\pi)|append((eq \cup \{b = [b]_\sigma\}, \{l : while(b), \sigma, \epsilon\}), \rho)}$$

The rule is the same as the former.

(3) *Equation Solver.* Find the first equation set $\rho[m]$ in the equation set list that has no solution. The statement block corresponded to the equation set is crucial. Statements from the corresponding statement block in $\rho[m]$ to $\rho[1]$ could be a minimal faulty program segment.

The equation set with no solution implies that under no state σ could the statements in [m] be executed to get the expected result.

(4) *Iteratively Minimize Segment.* The first-time locating segment might be too large for programmer to find out the real place of the faults. An iterative process need to be performed to make it easier for the programmer to locate faults. Programmer needs to give an expected value of a variable in the segment, and repeat from step 1 to minimize the faulty segment until code fragment is considered small enough for programmer to locate the cause of fault.

3 Bug Localization Framework

Intra-function debugging algorithm generates the locating equation for every statement in the program and then solve the equation set to get a faulty program segment. The algorithm performs better on computational faults than other kinds of faults (especially the control flow related faults). When the execution trace goes to a wrong path caused by some faulty predicates (branches in program), the algorithm may give an irrelevant program segment which could lead to a bad localization. It is obvious that BMPS could solve the control flow related fault, while Intra-function debugging algorithm is opposite. We propose an algorithm combined with the two methods, therefore it can deal with both the control flow related and compute related faults.

Firstly, we use Intra-function algorithm to give a coarse-grained faulty program segment according to the expected variable value given by programmer. It definitely reduce the space and the number of predicates need to be switched. Then we call BMPS to switch the outcomes of the predicates contained in the program segment. Also, programmer needs to check the limited number of statements in the segment to eliminate the compute related fault. One pass may solve one single bug. An iterative process need to be done if the program consists of multiple bugs. Programmer needs to give another expected value of a variable in the segment, and repeat the steps above to minimize the faulty segment step by step.

If the locating result turns out unsatisfiable, we may use BMPS for the whole program. As we said before, if the problem is caused by control flow faults

and the execution trace goes to the wrong path, the Intra-function debugging algorithm would give an irrelevant program segment leading to a bad localization result. This pass is aimed to deal with the program under such a situation. The disadvantage is that the full scale switch would consume much more time and space. More variables and paths would increase the number of variables and clauses in the SAT problem (satisfiability problem), which would make it more difficult and complex to solve the problem.

The overview of our algorithm is given in Fig. 3. The algorithm has 5 main steps.

Step 1: Find Faulty Value
When facing a failed run, inspect the input and output of test case to identify a proper variable with unexpected value.
Step 2: Reduce Faulty Program Segment
Run Intra-function debugging algorithm to get a minimal faulty program segment.
Step 3: Small Scale Predicate Switch
Run BMPS in the small segment of faulty program. Only the predicates in the segment is switchable, which means they are the candidates to be switched.
Step 4: Iterative Debugging and Locating
Repeatedly run step2 and step3 to get a more precise result in order to fix multiple bugs.
Step 5(Additional): Full Scale Predicate Switch
Run BMPS for the whole program if the results above seems to be unsatisfiable.

Fig. 3. Algorithm Overview.

4 Optimization

In the previous paper, the author proposed the prototype of bounded debugging via multiple predicate switching algorithm. It is an excellent algorithm, however, optimization could be made somewhere.

4.1 Use-Definition Chain

A Use-Definition Chain (UD Chain) is a sparse representation of data-flow information about variables [6]. A UD chain connects a use to all the definitions that may flow to it. Abstractly a UD chain is a function from a variable and a basic-block-position pair to sets of basic-block-position pairs, one for each use or definition, respectively. Concretely, they are generally represented by linked lists. They can be constructed by solving the reaching definitions data-flow problem for a procedure and then using the resulting information to build the linked list.

Making the UD chain is a step in liveness analysis, so that logical representations of all the variables can be identified and tracked through the code.

- *Use of Variable*. If variable, v, is on the RHS of statement $s(j)$, there is a statement $s(i)$ with $i < j$ and $min(j - i)$, that it is a definition of v and it has a use at $s(j)$ (or, in short, when a variable, v, is on the RHS of a statement $s(j)$, then v has a use at statement $s(j)$).

- *Definition of Variable*. When a variable, v, is on the LHS of an assignment statement, such as $s(j)$, then $s(j)$ is a definition of v. Every variable has at least one definition by its declaration (or initialization).

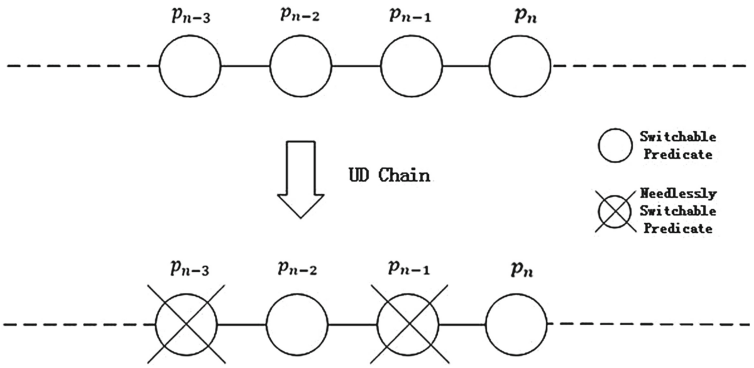

Fig. 4. Use-Definition Chain.

In out algorithm, we use UD chain to eliminate the redundant predicates need to be switched. When a test case failed, it means the result turned out wrong and the values of some variables were different from the expectation. When we get the UD chain of a variable, the statements which use the variable or related variable would be gathered. Of course, we can acquire the predicates in the statement sets. We treat these predicates as switchable and more suspicious ones than that are not in the sets.

As it is showed in Fig. 4, BMPS treats every predicate to be switchable. There are at least n predicates treated as candidates in the program. Then, BMPS may switch any of them to find out an execution path to get the right result. However, after the process of UD chain, we get a chain of predicates which is related to the variable with unexpected value. In another words, these predicates may influence the value of the variable with wrong value, and the faults may lie in these predicates. On the contrary, the other predicates may be unrelated to the variable and would not be needed to be switched so that we can save more time and space.

4.2 Predicate Switch Strategy

As we have already stated, in order to get the right results we have to switch conditional branch outcomes till we find some predicates which could be switched

to make the program produce the correct output. It is our goal to develop a switching strategy to make the algorithm more practical and effective.

- *Manually set limits.* Although we have implemented UD chain to the algorithm, some predicates could also be eliminated from the switchable predicate set. Programmer could mark some predicates to be not switched to make it more effective.

```
if(Predicate-1)
{
  ...
  if(Predicate-m)
  ...
}
else
{
  ...
  if(Predicate-n)
  ...
}
```

Fig. 5. Example of a Program.

The example in Fig. 5 is a small program. It contains an if-else block with various conditional branch in each sub-block. They are all predicates in switchable predicate set and may be switched to get right result. However, if it is certain that the execution path go into if part of the very first predicates (Predicate-1), it can be eliminated from the switchable predicate set. As for this case, it would at least reduce a half of the total spend in space and time. This action would make the algorithm more smart.

- *Increasingly number of predicates to be switched at a time.* Although we are aiming at the program with multiple bugs, sometimes only one predicate switch could lead to the right result and this may always happen in fact. Thus, we limit the number of predicates to be switched from one to all. After experiment, we found that most test case with control flow related bugs could be solved by only switching a small amount of predicates. Firstly, we only switch one predicate at a time, if we cannot get to the right result or the programmer are not satisfied with it, we increase the number of predicates to be switched at a time. Every time, we add the number by one. But, this may spend more time than the normal algorithm.

- *Last executed first switched ordering.* In the previous paper, Zhang and Gupta have presented that execution of faulty code (i.e. root cause of a failed run) is often not far away from the point at which the program fails (e.g., program crashes or it produces a wrong output value). Therefore, we can switch the predicates in reverse order. That is to say, the predicate that is closer to the faulty results (i.e. the end of the program in some situation) is earlier to be switched.

4.3 Upper Bound for Loops

In the previous paper, Liu proposed to set an upper bound for the iterative programs manually. That is to say, when programmer face with a program with loops (e.g., while, for statements), he must give BMPS a number to describe the times the loop circulates. Like the usual model checker methodology, we unwind the loop by duplicate the loop body n times, where n is the unwinding limit. However, it is difficult for programmer to give an accurate number for the iteration times. So we use Loopus [7] , a tool that automatically compute loop bounds for C programs, to give the upper bound of loop time.

5 Experiment and Evaluation

We implement BMPS by using CBMC, a bounded model checker for ANSI C programs (Clarke, Kroening, and Lerda 2004) [8,9]. For the implementation of Intra-function debugging algorithm, we use LLVM to parse the input program and generate the symbolic execution trace. We solve the CNF and equation set by implementing the Z3 (a SAT solver). All of our experiments are performed on a 3.10 GHZ Intel Core 2 Duo CPU with 4 GB RAM.

- *CBMC*. The tool produces a Boolean formula for a C program. We add assume and assert statements to give program specification. Then we use z3 (a SAT solver) [10] to check the formula generated by CBMC. If the formula is satisfiable, it means we generate a counter-example that meet the expected result. Let p_i be a predicate in the program, we replace p_i with the expression $sw_i?nondet\_bool(): p_i$. In the expression, sw_i is a Boolean variable and the function $nondet\_bool()$ returns a non-deterministic Boolean value. The expression controls the execution to go p_i or $\neg p_i$. Finally, We add $assume()$ to give the expected results of the variable; and $assert(0)$ to make the program stoppable.

- *Static Program Slice*. We use static data and control flow analysis to acquire a set of program statements related to the specified variable [11,12]. After predicate switch, we get an execution path where some predicate have already been switched. It is not apparent for the programmers to find out the localization of faults if we just show these switched predicate statement to them. Thus, we use static program slice to give programmer a set of statements that influence the switched predicates which may help to reveal the bugs.

5.1 Experiment Result

We use Siemens suite for experiment and evaluation. It consists of 7 base programs. Each of them have a number of faulty versions and test case. All the programs (implemented in C language), test cases and defect data can be downloaded from the website http://www-static.cc.gatech.edu/aristotle/Tools/subjects at Georgia Tech. We used TCAS program and REPLACE program in the Siemens suite. The TCAS program constitutes an aircraft collision avoidance system with 173 lines of code and 41 faulty versions.

Table 1. Results of running our algorithm on TCAS

Version	#ErrNum	DetectReduce	RunTime	ErrorType
v1	1	96	0.26	Control Flow
V2	1	95	0.24	Control Flow
V3	1	98	0.28	Control Flow
V4	1	96	0.29	Control Flow
V5	1	72	0.55	Compute
V6	1	97	0.31	Control Flow
V7	1	78	0.56	Compute
V8	1	79	0.49	Compute
V9	1	96	0.31	Control Flow
V10	2	98	0.29	Control Flow
V11	2	98	0.28	Control Flow
V12	1	97	0.25	Control Flow
V13	1	71	0.47	Compute
V14	1	75	0.44	Compute
V15	3	69	0.45	Compute
V16	1	72	0.49	Compute
V17	1	75	0.50	Compute
V18	1	77	0.41	Compute
V19	1	78	0.47	Compute
V20	1	95	0.26	Control Flow
V21	1	96	0.28	Control Flow
V22	1	98	0.25	Control Flow
V23	1	96	0.29	Control Flow
V24	1	97	0.26	Control Flow
V25	1	97	0.30	Control Flow
V26	1	79	0.50	Add Code
V27	1	74	0.52	Add Code
V28	1	95	0.30	Control Flow
V29	1	96	0.28	Control Flow
V30	1	95	0.29	Control Flow
V31	2	70	0.51	Add Code
V32	2	71	0.52	Add Code
V33	4	–	–	Crash
V34	1	97	0.31	Control Flow
V35	1	98	0.27	Control Flow
V36	1	78	0.47	Compute
V37	1	97	0.29	Control Flow
V38	1	–	–	Crash
V39	1	98	0.29	Control Flow
V40	2	97	0.31	Control Flow
V41	1	96	0.28	Control Flow

Table 1 below shows the result of running our algorithm on TCAS. **#ErrNum** is the total number of bugs in program, **DetectReduce** means the reduction of code needs to be checked after running our algorithm, **RunTime** is the running time of algorithm, **ErrorType** is the type of error in a program. Among the 41 versions, 4 versions add some codes to the original version, 2 versions crash, 11 versions are compute related bugs, the others are control flow related bugs. The experiment result shows that our approach deals well with the versions containing control flow related bugs. It reduces at least 95 % of the detection scope, because we use modified BMPS. For the general control flow related bugs, it can figure out a set of predicates which could be switched to get to the expected result. The number of predicates in the set is not more than 10 as usual. After examining the predicates, the exact localizations of bugs are detected. As for the compute related bugs, our approach gives a minimal suspicious program segment. It reduces 75 % of the detection scope averagely. Furthermore, these program segments are context-sensitive which could help the programmer to understand the cause of bugs.

As for the run time of the algorithm, it runs about 0.3 s for the program with control flow related bugs averagely and 0.51 s for those with compute related bugs. The run time of the former ones are more than the latter ones, because we run an additional full scale predicate switch if the small scale predicate switch cannot find a critical predicate set.

Obviously, our approach is better than BMPS. Firstly, it cannot deal with the compute related bugs. Secondly, BMPS may give the programmer an irrelevant predicate to reveal the bugs. For example, Fig. 6 below is a segment of TCAS. The output of TCAS is the variable *alt_sep*, and a conditional statement is placed at the end of the program, which means after executing this conditional statements the execution of TCAS is over. Suppose that *alt_sep* is expected to be $UNRESOLVED$, but the value is $UPWARD\_RA$ for some reason. Thus, there are some bugs in program. But, when using BMPS, we could always switch the predicates in this conditional statement to get the right result which would absolutely conceal the real bugs. In our approach, programmer can mark this conditional statement to be errorless manually, then the algorithm would find out the real locations of bugs.

```
if (need_upward_RA && need_downward_RA)
    alt_sep = UNRESOLVED;
else if (need_upward_RA)
    alt_sep = UPWARD_RA;
else if(need_downaward_RA)
    alt_sep = DOWNWARD_RA;

return alt_sep;
```

Fig. 6. Example of TCAS.

As for the REPLACE program, it has 564 lines of code, and 32 faulty versions. Among the 32 versions, 2 versions miss code, 2 versions add code, 19 versions are control flow related faults. It has many loops, our algorithm could effectively reduce the number of candidate switchable predicates, which could apparently reduce the total running time. But we cannot deal with the situation that code are missing.

Also, we intentionally combine the control flow bug related version with the other version to form some new versions with multiple bugs for TCAS program. Our approach could deal with these situation in about 4 iterative processes averagely. The run time could not be counted, for the process is an iterative subprocedures with programmer.

6 Related Work

In a previous paper, the author (Zhang, Gupta and Gupta) firstly proposed an algorithm to locate faults through automated predicate switch. They demonstrated that by forcibly switching the outcome of a predicate at runtime the program state can be modified. But they only switch one single predicate at a time. In 2010, Liu and Li presented a first step towards a theoretical exploration of program debugging algorithm (BMPS). They switch multiple predicates at a time in order to get a successful execution result.

Griesmayer et al. [13] gives a fault localization algorithm for C programs by constructing a modified program that allows a given number of expressions to be changed arbitrarily. They use the counterexample trace from a Model Checker (CBMC). Because every expression could be modified, the search space is extremely large. Like what we did in our approach, for an expression e, they introduce a Boolean variable sw and replace e with $sw?nondet() : e$. But, our search space is smaller than theirs for control flow related problem.

BugAssist [14] minimizes a given error trace obtained from bounded model checking using a Max-SAT algorithm. When a test case failed, BugAssist generates an execution trace and transfers it into an unsatisfiable formula combined with the input and the assertion. Thus, it has an apparent limitation that only executable statements can be part of the minimized error trace, which means only the bugs in the executable statements could be revealed. They treat bug locating problem as a partial maximum satisfiability problem which asks what is the maximum number of clauses that can be satisfied by any assignment. However, bug locating is a SAT problem in our paper. Also, we can locate bugs in the statement that is not executed under specified test case.

Many bug locating work uses the difference and comparison between the error trace and successful trace. Groces approach [15] calls CBMC to get a failing run firstly, then computes the difference between a failing run and a closest correct run. Ball et al. [16] call a model checker several times and compares the counterexamples to a successful trace. They believe that the faults are those transitions that do not appear in a correct trace. Our approach does not need to compare the successful trace and the error trace, thus we need less information.

For the control flow related bug, our approach gives an exact location of the bug instead of a code fragment.

7 Conclusion

In this paper, based on Liu et al.'s work on automated debugging via multiple predicate switching, we proposed an algorithm combined BMPS with Intra-function debugging method. So it would not only deal with the control flow related bugs, but also the compute related bugs. We optimize the algorithm by Use-Define chain, strategy for predicate switching and automated upper bound for loops. The search space for our approach is much reduced compared with the former algorithm. The results turned out that it is promising and more efficient.

8 Future Work

In the furture, we would like to extend the algorithm to fit pointer better. Moreover, a full implementation should be developed, and sufficient experiments should be made.

References

1. Liu, Y., Li, B.: Automated program debugging via multiple predicate switching. In: AAAI (2010)
2. Wong, W.E., Debroy, V.: A survey of software fault localization. Department of Computer Science, University of Texas at Dallas, Technical report UTDCS-45 9 (2009)
3. Cleve, H., Zeller, A.: Locating causes of program failures. In: Proceedings of the 27th International Conference on Software Engineering, pp. 342–351. ACM (2005)
4. Zhang, X., Gupta, N., Gupta, R.: Locating faults through automated predicate switching. In: Proceedings of the 28th International Conference on Software Engineering, pp. 272–281. ACM (2006)
5. Challenge, D.: Satisfiability: Suggested format. DIMACS Challenge, DIMACS (1993)
6. Muchnick, S.S.: Advanced Compiler Design Implementation. Morgan Kaufmann, San Francisco (1997)
7. Sinn, M., Zuleger, F.: Loopus-a tool for computing loop bounds for c programs. In: WING@ ETAPS/IJCAR, Citeseer, pp. 185–186 (2010)
8. Clarke, E., Kroning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
9. Clarke, E., Kroening, D., Yorav, K.: Behavioral consistency of c and verilog programs using bounded model checking. In: Design Automation Conference, Proceedings, pp. 368–371. IEEE (2003)
10. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)

11. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, pp. 439–449. IEEE Press (1981)
12. Agrawal, H., Horgan, J.R.: Dynamic program slicing. In: ACM SIGPLAN Notices, vol. 25, 246–256. ACM (1990)
13. Griesmayer, A., Staber, S., Bloem, R.: Automated fault localization for C programs. Electron. Notes Theoret. Comput. Sci. **174**, 95–111 (2007)
14. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. ACM SIGPLAN Notices **46**, 437–446 (2011)
15. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. Int. J. Soft. Tools Technol. Transf. **8**, 229–247 (2006)
16. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: ACM SIGPLAN Notices. vol. 38, pp. 97–105. ACM (2003)