Automated Program Debugging via Multiple Predicate Switching

Yongmei Liu and **Bing Li**
Department of Computer Science
Sun Yat-sen University
Guangzhou 510006, China
ymliu@mail.sysu.edu.cn, libing5@mail2.sysu.edu.cn

Abstract

In a previous paper, Liu argued for the importance of establishing a precise theoretical foundation for program debugging from first principles. In this paper, we present a first step towards a theoretical exploration of program debugging algorithms. The starting point of our work is the recent debugging approach based on predicate switching. The idea is to switch the outcome of an instance of a predicate to bring the program execution to a successful completion and then identify the fault by examining the switched predicate. However, no theoretical analysis of the approach is available. In this paper, we generalize the above idea, and propose the bounded debugging via multiple predicate switching (BMPS) algorithm, which 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. Clearly, BMPS can be implemented by resorting to a SAT solver. We focus attention on RHS faults, that is, faults that occur in the control predicates and right-hand-sides of assignment statements. We prove that for conditional programs, BMPS is quasi-complete for RHS faults in the sense that some part of any true diagnosis will be returned by BMPS; and for iterative programs, when the bound is sufficiently large, BMPS is also quasi-complete for RHS faults. Initial experimentation with debugging small C programs showed that BMPS can quickly and effectively locate the faults.

Introduction

Program debugging is one of the most time-consuming parts of the software development cycle. In recent years, automatic debugging has been an active research area in software engineering. However, existing approaches are mostly experiential, that is, they depend on expert experience and heuristic information. In a previous paper, Liu (2008) argued for the importance of establishing a precise theoretical foundation for debugging from first principles, which would include two ingredients: a formal definition of the debugging task, and an exploration of the associated computational problem. Liu gave a formalization of the program debugging task in the situation calculus, a logical language suitable for describing dynamic worlds. In this paper, we present a first step towards a theoretical exploration of debugging algorithms.

A general approach to automated debugging is based on modifying the program state to bring the execution to a successful completion. However, searching for arbitrary state changes is difficult due to the extremely large search space. A recent solution proposed by (Zhang, Gupta, and Gupta 2006) is to only switch the outcome of an instance of a predicate and then identify the fault by examining the switched predicate, called *critical predicate*. Clearly, the search space for predicate switching is far less than that for arbitrary state changes. Through experimental evaluation, the authors found their approach to be practical and effective. However, they didn't give any theoretical analysis of their approach, for example, they didn't analyze under what situations their approach is applicable. Obviously, in some situations, a single predicate switch is not sufficient.

In this paper, we generalize the idea behind the above approach, and propose the bounded debugging via multiple predicate switching (BMPS) algorithm, which 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. Clearly, BMPS can be implemented by resorting to a SAT solver. As in (Liu 2008), we restrict our attention to C-like programs without procedures, called *while programs*. A formal study of program debugging has to resort to the formal semantics of programs. As in (Liu 2008), we treat while programs as Golog programs, and hence obtain formal semantics of while programs in the situation calculus via the semantics of Golog, a programming language for high-level robotic control (Levesque et al. 1997).

The key concept underlying our approach is that of *critical predicate sets*. Intuitively, a critical predicate set for a failing test case is a set of predicates whose outcomes we can switch at runtime to produce the correct output. In this paper, we restrict our attention to *RHS faults*, that is, faults that occur in control predicates or right-hand-sides of assignment statements. By applying restrictions on the *program dependency graph*, we identify a class of faults which we call *predicate-cut faults*. Intuitively, for predicate-cut faults, the only way errors propagate is through control predicates. Based on a key property which relates critical predicate sets to predicate-cut faults, we prove that for conditional programs, BMPS is *quasi-complete* for RHS faults in the sense that some part of any true diagnosis will be returned by

BMPS; and for iterative programs, when the bound is sufficiently large, BMPS is also quasi-complete.

We experimented with implementing BMPS by using CBMC (Clarke, Kroening, and Lerda 2004), a bounded model checker for C programs. Our experiments with debugging a dozen of programs written for C programming exercises showed that BMPS can quickly and effectively locate the faults. We also analyzed the TCAS task of the *Siemens Suite* (Rothermel and Harrold 1999), which has been used as a benchmark suite for debugging approaches. Among the 41 faulty versions of the TCAS program, 39 versions are RHS faults, among which only one is non-predicate-cut fault.

Background Work

The Situation Calculus and Golog

The situation calculus (Reiter 2001) is a many-sorted first-order language (with some second-order ingredients) suitable for describing dynamic worlds. There are three disjoint sorts: $action$ for actions, $situation$ for situations, and $object$ for everything else. A situation calculus language $\mathcal{L}$ has the following components: a constant S_0 denoting the initial situation; a binary function $do(a, s)$ denoting the successor situation to s resulting from performing action a; a binary predicate $Poss(a, s)$ meaning that action a is possible in situation s; action functions, ***e.g.***, $move(x, y)$; relational fluents, ***i.e.***, predicates taking a situation term as their last argument, ***e.g.***, $ontable(x, s)$; functional fluents, ***e.g.***, $height(x, s)$; and situation-independent predicates and functions. We use $\mathcal{L}^-$ to denote the language obtained from $\mathcal{L}$ by removing the sort $situation$ and removing the situation argument from every fluent. We call an $\mathcal{L}^-$-formula a *pseudo-fluent* formula. Let ϕ be such a formula, and s be a situation term. We let $\phi[s]$ denote the formula obtained from ϕ by restoring s as the situation arguments to all fluents.

The formal semantics of Golog is specified by an abbreviation $Do(\delta, s, s')$, which is inductively defined as follows:

1. Primitive actions: For any action term α,
   $Do(\alpha, s, s') \overset{def}{=} Poss(\alpha, s) \wedge s' = do(\alpha, s)$.
2. Test actions: For any pseudo-fluent formula ϕ,
   $Do(\phi?, s, s') \overset{def}{=} \phi[s] \wedge s = s'$.
3. Sequence:
   $Do(\delta_1; \delta_2, s, s') \overset{def}{=} (\exists s'').Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s')$.
4. Nondeterministic choice of two actions:
   $Do(\delta_1 \,|\, \delta_2, s, s') \overset{def}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$.
5. Nondeterministic iteration:
   $Do(\delta^*, s, s') \overset{def}{=} (\forall R).\{(\forall s_1)R(s_1, s_1) \wedge (\forall s_1, s_2, s_3)$ $[R(s_1, s_2) \wedge Do(\delta, s_2, s_3) \supset R(s_1, s_3)]\} \supset R(s, s')$.
   The definition appeals to second-order logic to say that the relation represented by $Do(\delta^*, s, s')$ is the transitive closure of that by $Do(\delta, s, s')$.

We omit the definition of other constructs such as procedures. Conditionals and loops are defined as abbreviations:

$$\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ fi} \overset{def}{=} [\phi?; \delta_1] \,|\, [\neg\phi?; \delta_2],$$
$$\textbf{while } \phi \textbf{ do } \delta \textbf{ od} \overset{def}{=} [\phi?; \delta]^*; \neg\phi?.$$

Formalization of Program Debugging

Liu (2008) presented a formalization of the program debugging task in the situation calculus. The author restricted attention to the debugging of *while* programs. We make the same restriction in this paper. Let L_P be the language of Peano Arithmetic, ***i.e.***, $L_P = \{0, s, +, \cdot, =\}$, where "s" denotes the successor function. Let L be L_P extended with the minus, division, modulo, and comparison operations, ***i.e.***, $L = L_P \cup \{-, /, \%, <, <=\}$. We call quantifier-free formulas of L *conditions*, and formulas of L *assertions*. We let $\mathcal{W}$ denote the least set of programs such that

1. for every variable x and term t of L, $x := t \in \mathcal{W}$;
2. if $\delta_1, \delta_2 \in \mathcal{W}$, then $\delta_1; \delta_2 \in \mathcal{W}$.
3. if $\delta_1, \delta_2 \in \mathcal{W}$, then for every condition e,
   if e **then** δ_1 **else** δ_2 **fi** $\in \mathcal{W}$;
4. if $\delta \in \mathcal{W}$, then for every condition e,
   while e **do** δ **od** $\in \mathcal{W}$.

We call the condition e in an if or while statement a *control predicate*. In this paper, by *components* of a while program P, we mean the set of assignment statements and control predicates of P. We distinguish between a component and its expression: two different assignment (resp. predicate) components might have the same expression.

Liu (2008) treated while programs as Golog programs and hence obtain the formal semantics of while programs in the situation calculus. The situation calculus language of arithmetic programming $\mathcal{L}_{ap}$ contains the following symbols:

1. An infinite set of memory location constants $L_1, L_2, \ldots$;
2. A set of operator constants such as Add and Minus;
3. A functional fluent $val(x, s)$ which denotes the value of memory location x in situation s;
4. Action $assn(z, o, x, y)$, which assigns to memory location z the result of applying operator o to the values of memory locations x and y.

Clearly, each assignment statement can be represented as a sequence of $assn$ actions. Hence each while program can be expressed as a Golog program whose primitive actions are $assn$ actions.

The basic action theory of arithmetic programming $\mathcal{D}_{ap}$ consists of the following:

1. The foundational axioms for situations;
2. The action precondition axiom for $assn$;
3. The successor state axiom for $assn$ which states that the execution of $assn(z, o, x, y)$ modifies the value of memory location z according to the arguments o, x, and y, and leave unchanged the values of the other locations;
4. Unique names axioms for $assn$;
5. Initial database which includes the second-order axiomatization of Peano arithmetic.

Liu (2008) defined test cases as follows: A *test case* T for a program P is a pair $\langle In_T, Out_T\rangle$ of assertions where In_T specifies the exact values of the input variables, and Out_T specifies the values of the output variables.

Note that a while program is a deterministic one. Hence a while program P passes a test case T iff the formula $Do(In_T?; P; Out_T?, s, s')$ is satisfiable wrt the theory $\mathcal{D}_{ap}$.

Basic Concepts

In this section, we define the basic concepts we use in this paper and present their properties.

Execution Paths and Traces

When introducing Golog, we defined conditionals and loops as abbreviations. Now we formally define the concept of execution paths and traces for while programs.

Definition 1 Let P be a while program. We do the following operations on its Golog representation:

1. replace each δ^* with $\epsilon|\delta|\delta^2|\ldots|\delta^n|\ldots$, where ϵ denotes the empty program;
2. repeatedly apply the following distribution laws until they are not applicable:
   $\delta;(\delta_1|\delta_2) \Leftrightarrow (\delta;\delta_1)|(\delta;\delta_2),\ \ (\delta_1|\delta_2);\delta \Leftrightarrow (\delta_1;\delta)|(\delta_2;\delta)$.

The result is a nondeterministic choice of (possibly infinitely many) sequential programs consisting of assignments and tests. We call each of them an *execution path* for P, and we denote by EP(P) the set of all execution paths for P. We call the sequence of assignments contained in an execution path β the *execution trace* for β.

An execution path β carries with it the conditions for its trace to be executed. For each test t on β, it is either a positive test $e?$ or a negative test $\neg e?$ of some predicate e of P; after the sequence of assignments that occur before t has been executed, the truth value of e must be the same as the polarity of the test.

For example, below is a program POWER which intends to compute the nth power of 2 and its two execution paths:

```
p = 1;  i = 1;
while (i < n)
{  p = p * 2; i = i + 1;  }
```

$\beta_1 : p = 1; i = 1; \neg(i < n)?.$
$\beta_2 : p = 1; i = 1; (i < n)?; p = p * 2; i = i + 1; \neg(i < n)?.$

Proposition 1

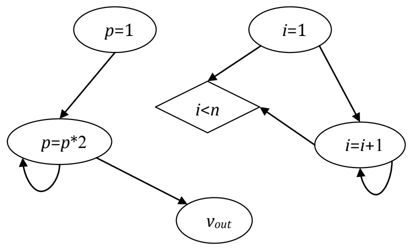
1. *EP*(δ) $= \{\delta\}$*, where δ is an assignment or a test;*
2. *EP*($\delta_1;\delta_2$) $= EP(\delta_1); EP(\delta_2)$*, which denotes the set* $\{\beta_1;\beta_2 \mid \beta_1 \in EP(\delta_1), \beta_2 \in EP(\delta_2)\}$*;*
3. *EP*($\delta_1|\delta_2$) $=$ *EP*(δ_1) $\cup$ *EP*(δ_2).

Data Dependency Graphs

The dependency graph for a program describes the data dependency and control dependency among statements of the program. In this paper, we only need to use data dependency. We now formally define the concept of data dependency graph for a program.

Definition 2 Let P be a while program. The *data dependency graph* (DDG) for P is a directed graph. The vertices are components of P together with a special output vertex v_{out} which uses all the output variables. There is an edge from v_1 to v_2 iff there is an execution path β for P such that v_1 occurs before v_2, v_2 uses a variable assigned by v_1, and there is no re-assignment of the variable between v_1 and v_2.

The following figure shows the DDG for POWER.



An important concept we use in our algorithm is that of backward data slice.

Definition 3 Let G be the DDG of a program, and let v be a node. The *backward data slice* of v, denoted BDS(v), is the set of nodes from which v is reachable. The backward data slice of a set V of nodes is the union of the backward data slices of nodes in V.

Note the difference between backward data slice and the concept of *backward slice* in the literature. The backward slice of a node v consists of nodes from which v is reachable through a path of both data and control dependency. An experimental study by (Zhang, Gupta, and Gupta 2006) showed that the backward data slice of a node is much smaller than the backward slice.

In the above example, the backward data slice of the predicate $i < n$ is $\{i = 1, i = i + 1, i < n\}$.

Diagnoses

A commonly used evaluation measure for debugging approaches is the scoring function proposed by Renieris and Reiss (2003). This measure assumes the existence of a correct program; the differences between the correct program and its faulty version point to where the fault is. In this paper, we make a similar assumption: there are possibly multiple correct programs obtained by replacing one or more assignments and/or predicates in the faulty program.

Definition 4 A *substitution function* θ for a program P is a mapping from the components of P to assignment and predicate expressions. We use $\theta(P)$ to denote the program obtained from P by applying θ. To overload the notation, for a component v of P, we also use $\theta(v)$ to represent the corresponding component of the program $\theta(P)$.

Definition 5 Let P be a faulty program, and θ a substitution function for P such that $\theta(P)$ is a correct version of P. We call θ a *repairing function* for P. We call the set of components of P whose expressions are modified by θ a *diagnosis* for P, and we say components in Δ are faulty wrt θ.

Definition 6 Let P be a faulty program, θ a repairing function for P and Δ its associated diagnosis. We call Δ an *RHS diagnosis* or an *RHS fault* if for each assignment α of P, α and $\theta(\alpha)$ only differ on the right-hand-side (RHS). We call Δ a *predicate-cut diagnosis* or a *predicate-cut fault* if it is an RHS diagnosis and in the DDG of P, there is no path from a node in Δ to the output node.

For example, there are two diagnoses $\{i = 1\}$ and $\{i < n\}$ for POWER, since we can either replace $i = 1$ with $i = 0$ or replace $i < n$ with $i \le n$, and get a correct program. From the DDG, it is easy to see that both are predicate-cut diagnoses. In contrast, for the following program POWER′, there is one diagnosis $\{p = p+2\}$, which is a non-predicate-cut RHS diagnosis.

```
p = 1;  i = 0;
while (i < n)
{  p = p + 2; i = i + 1;  }
```

RHS diagnoses have the following property:

Proposition 2 *Let P be a program, Δ an RHS diagnosis and θ its repairing function. Let $v \notin \Delta$. Then there is a path from Δ to v iff there is a path from $\theta(\Delta)$ to $\theta(v)$.*

The intuitive idea behind predicate-cut faults is that the only way errors propagate is through control predicates. By the above proposition, we have:

Proposition 3 *Let P be a program, Δ a non-predicate-cut RHS diagnosis. Then $\Delta \cap BDS(v_{out}) \neq \emptyset$.*

Thus any non-predicate-cut RHS fault is captured by BDS(v_{out}).

Critical Predicate Sets

Recall the definition of test cases from the section on background work. For example, POWER has a correct test case $\langle n = 0, p = 1\rangle$, and two failing test cases $\langle n = 1, p = 2\rangle$ and $\langle n = 5, p = 32\rangle$.

Definition 7 Let τ be a sequence of assignments. We call τ a *correct execution trace* for a test case T if the execution of τ produces the correct output for T, more formally, if $Do(In_T?; \tau; Out_T?, s, s')$ is satisfiable wrt $\mathcal{D}_{ap}$.

Definition 8 Let P be a while program, and T a failing test case for P. Let β be a correct execution path of P for T, that is, its trace is a correct trace for T. Let C be the set of predicates e for which there exists a positive/negative test t of e on β such that the polarity of t is different from the truth value of e after the trace segment before t has been executed for T. We call C a *critical predicate set* (CPS) wrt T induced by β. If each loop is executed at most k times on β, we call C a depth k CPS.

Intuitively, a CPS wrt T induced by β is the set of predicates e such that we have to switch the outcome of some instance of e in order for the execution of T to take the path β. Of course, for a test case T, there might be multiple correct execution paths for it; hence there might be multiple CPSes wrt T, and we can define the concept of minimal CPS.

We now prove an important property of critical predicate sets, which essentially says that any predicate-cut fault is captured by the BDS of any predicate in some CPS. We need the proposition below, which follows from Proposition 1.

Proposition 4 *Let P be a program, and θ a substitution function for P. Then EP$(\theta(P)) = \theta($EP$(P))$.*

Proposition 5 *Let P be a while program, Δ a predicate-cut diagnosis for P, and T a failing test case for P. Then there exists a CPS C such that for each $e \in C$, $\Delta \cap BDS(e) \neq \emptyset$.*

Proof: Let θ be the repairing function associated with Δ. By Proposition 4, EP$(\theta(P)) = \theta($EP$(P))$. Let β be an execution path for P such that $\theta(\beta)$ is the execution path of $\theta(P)$ for test case T. Let τ and τ' be the traces of β and $\theta(\beta)$, respectively. We claim that they have the same effect on the output variables. It suffices to prove that there is no path from Δ to the output node, and in the DDG of $\theta(P)$, there is no path from $\theta(\Delta)$ to the output node either. This holds by Proposition 2, since Δ is a predicate-cut diagnosis. Hence τ is a correct execution trace for T.

Now let C be the CPS induced by β. Let $e \in C$. If $e \in \Delta$, we have that $\Delta \cap$ BDS$(e) \neq \emptyset$. So assume that $e \notin \Delta$. By the definition of CPS, there exists a test t of e on β such that the polarity of t is different from the truth value of e after the trace segment before t, which we denote by γ, has been executed for T. Since $\theta(\beta)$ is the execution path of $\theta(P)$ for T, the polarity of t is the same as the truth value of e after $\theta(\gamma)$ has been executed for T. So the truth value of e after γ is executed for T is different from that after $\theta(\gamma)$ is executed for T. Thus either there is a path from Δ to e, or there is a path from $\theta(\Delta)$ to e. Since Δ is a predicate-cut diagnosis, by Proposition 2, in either case, there is a path from Δ to e. Hence, $\Delta \cap$ BDS$(e) \neq \emptyset$. ■

The Bounded Debugging via Multiple Predicate Switching Algorithm

In this section, we first consider debugging of conditional programs, and then debugging of iterative programs. Finally, we discuss an extension of the algorithm.

Conditional Programs

Since there are no loops in conditional programs, critical predicate sets are computable.

Proposition 6 *Let P be a conditional program, and T a failing test case for P. We can construct a Boolean formula A such that its models correspond to the CPSes of P wrt T. The size of A is linear in the size of P.*

Proof: We obtain a program P' from P as follows: for each predicate e in P, we introduce a Boolean variable sw_e, and replace e with $sw_e ? \neg e : e$, which abbreviates the formula $\neg sw_e \wedge e \vee sw_e \wedge \neg e$. We then construct a Boolean formula A encoding the executions of the program $In_T?; P'; Out_T?$, say using the encoding method of CBMC (see the section on experimentation and evaluation). Then for any truth assignment σ of sw_e's, σ satisfies A iff the set $\{e \mid \sigma(sw_e) = 1\}$ is a CPS. ■

Thus we can use a SAT solver to compute all the minimal critical predicate sets. We get the following debugging via multiple predicate switching (MPS) algorithm and theorem.

MPS(P,T)
Input: a conditional program P, and a failing test case T
Output: a collection of sets of components of P

1. Output BDS(v_{out});
2. Compute all the minimal critical predicate sets;
3. For each minimal CPS C, output BDS(C).

Theorem 7 *Let P be a conditional program, and T a failing test case for P. For any RHS diagnosis Δ of P, there exists a set S returned by MPS such that $S \cap \Delta \neq \emptyset$.*

Proof: If Δ is not a predicate-cut diagnosis, by Proposition 3, $\Delta \cap \text{BDS}(v_{out}) \neq \emptyset$. Otherwise, by Proposition 5, there is a CPS C s.t. for each $e \in C$, $\Delta \cap \text{BDS}(e) \neq \emptyset$. Let C' be a minimal CPS contained in C. Then $\Delta \cap \text{BDS}(C') \neq \emptyset$. ■

Liu (2008) defined the concept of completeness for a debugging algorithm: it is complete if any diagnosis is a subset of some set returned by it. Thus essentially, the above theorem means that MPS is *quasi-complete* for RHS diagnoses. In fact, if we do not want a debugging algorithm to return many false diagnoses, the goal of completeness as defined by Liu (2008) is difficult to achieve.

Iterative Programs

For iterative programs, critical predicate sets should be uncomputable. To address this problem, like bounded model checking, we consider depth k critical predicate sets. Recall that a depth k CPS is one induced by an execution path where each loop is executed at most k times.

Definition 9 Let P be an iterative program, and let $k \in \mathbb{N}$. The unwinding of P to depth k, denoted P^k, is obtained from P as follows: replace each δ^* with $\epsilon|\delta|\delta^2|\ldots|\delta^k$.

Proposition 8 *Let P be an iterative program, T a failing test case for P, and $k \in \mathbb{N}$. We can construct a Boolean formula A such that its models correspond to the depth k CPSes of P wrt T. The size of A is linear in the size of P^k.*

Proof: We obtain a program P' from P^k as follows: for each predicate e in P, introduce a Boolean variable sw_e; for each instance ei of e in P^k, introduce a Boolean variable sw_{ei} and replace ei with $sw_{ei}?\neg e : e$. We then construct a Boolean formula B encoding the executions of the program $In_T?; P'; Out_T?$. Take A as the conjunction of B and the constraints $sw_e \equiv \bigvee sw_{ei}$, where the disjunction ranges over all instances of e, and e ranges over all predicates. ■

Thus we obtain the following bounded debugging via multiple predicate switching (BMPS) algorithm and theorem.

BMPS$_k$(P, T), where $k \in \mathbb{N}$
Input: an iterative program P, a failing test case T
Output: a collection of sets of components of P

```
1. Output BDS($v_{out}$);
2. Compute all the minimal depth $k$ critical predicate sets;
3. For each minimal depth $k$ CPS $C$, output BDS($C$).
```

Theorem 9 *Let P be an iterative program, and T a failing test case for P. Then there exists a k such that for any RHS diagnosis Δ of P, there exists a set S returned by BMPS$_k$ such that $\Delta \cap S \neq \emptyset$.*

Thus when the bound is sufficiently large, MBPS is quasi-complete for RHS diagnoses.

An Extension of the Algorithm

Clearly, we can extend the BMPS algorithm to also altering the outcome of assignment statements where the assigned variable is of an enumeration type with a small range of possible values. We call a predicate or such an assignment a *small-range-component* (SRC). We can generalize the concept of critical predicate set to that of *critical SRC set*. Also, we say that Δ is a *SRC-cut diagnosis* if there is an SRC on each path from a node in Δ to the output node.

Experimentation and Evaluation

The key part of the BMPS algorithm is to compute depth k critical predicate sets. We experimented with implementing this part by using CBMC, a bounded model checker for ANSI C programs (Clarke, Kroening, and Lerda 2004). CBMC supports assume and assert statements which can be used to give program specification. Given an ANSI C program P and an unwinding depth k, CBMC produces a Boolean formula that is satisfiable iff there is an execution of P where each loop is executed at most k times and which satisfies all assume statements and violates an assert statement. The formula is then checked by using a SAT solver. If the formula is satisfiable, a counter-example is extracted from the output of the SAT solver.

Given a while program P, a failing test case T, and $m \in \mathbb{N}$, we produce a program $\Pi(P, T, m)$ as follows. Here m is a bound on the size of critical predicate sets.

- Let $e_0, e_1, \ldots, e_{n-1}$ be all the predicates in P. We declare a Boolean array $sw[n]$, and replace each e_i with $sw[i]?nondet\_bool() : e_i$, where $nondet\_bool()$ returns a non-deterministic Boolean value.
- We set the values of the input variables according to T.
- We add $assume(Out_T)$; $assume(\Sigma_i sw[i] \leq m)$; $assert(false)$ at the end of the program.

Then we call CBMC on $\Pi(P, T, m)$ with bound k. When a counter-example is generated, we get a depth k CPS of size $\leq m$. With some additional manipulation, we can compute all minimal depth k critical predicate sets of a bounded size.

For example, let P be POWER, and T be the test case $\langle n = 5, p = 32\rangle$. Then below is the program $\Pi(P, T, 1)$:

```
void main() {
  int n,p,i;  _Bool sw;
  n = 5;  p = 1;  i = 1;
  while (sw?nondet_bool():i<n)
  {  p = p*2;  i = i+1;  }
  assume(p==32);  assert(0);  }
```

When we call CBMC with bound 6 on the above program, we obtain $i < n$ as a switched predicate. The running time is less than 0.1 second. The backward data slice of the predicate contains both true diagnoses $\{i = 1\}$ and $\{i < n\}$.

We experimented with debugging a dozen of while programs written for C programming exercises, such as computing the greatest common divisor or least common multiple of two numbers, computing the number of primes or Armstrong numbers within a certain range, etc. The results showed that BMPS can quickly and effectively locate the faults. In each case, the running time is less than 1 second. For example, the following is a program SORT for selection sort, where there are two faults.

```
void sort(int a[], int N) {
   int i,j,k,temp;
   for (i=0; i<N; i++)
   {  k = i;
      for (j=i+1; j<N; j++)
         if (a[j]>a[k]) k=j;
         // (correct version: a[j]<a[k])
```

```
        if (k<i) // (correct version: k!=i)
        {  temp = a[i]; a[i] = a[k];
           a[k] = temp; }}}
```

When we call CBMC on $\Pi(Sort, T, 2)$ with bound 6, where T is a test case of array of size 6, we get a CPS $\{a[j] > a[k], k < i\}$, which is exactly the true diagnosis.

An evaluation benchmark suite for debugging approaches is the *Siemens Suite* (Rothermel and Harrold 1999), which consists of 7 base programs, and for each of them, a number of faulty variations and a large number of test cases. Each variation is obtained by manually seeding the base program with faults, usually by modifying a single line of code. The TCAS program of the suite is used for aircraft conflict detection and resolution. It has 173 lines of code, no loops, and there are 41 faulty versions for it. The major part of the program consists of 3 functions: alt_sep_test, Non_Crossing_Biased_Climb, and Non_Crossing_Biased_Descend. Among the 41 versions, only two of them are not RHS faults: one is obtained from the correct version by deletion of an else branch, the other is obtained by modifying the declaration of an array variable. In the two Non_Crossing functions, all variables are Boolean, thus all assignments can be treated as predicates. In the alt_sep_test function, there are 4 predicates. By an manual examination, we identified among the 39 RHS faulty versions, only one is not predicate-cut: here the value for a constant is modified, and this constant is used in deciding the return value. We have run CBMC on many faulty versions of TCAS and their failing test cases, in each case, we get a CPS of size 1 in less than 0.01 second.

Related Work

In this paper, we resort to a SAT solver for program debugging. Two other approaches along this line are those of (Groce 2004) and (Griesmayer, Staber, and Bloem 2006). The first approach consists of the steps: call CBMC to get a failing run, use a pseudo-Boolean solver to get a closest correct run, and then compute the differences between the two runs. Given a failing run, the second approach reports program components that may be changed to avoid the failure. It first constructs a modified program that allows a given number of expressions (*i.e.*, control predicates and the right-hand-sides of assignments) to be changed arbitrarily and contains the negated specification from the original program. Then it calls CBMC to find a failing run for the new program. To construct the new program, for each expression e_i, introduce a Boolean variable ab_i which represents that e_i is abnormal, and replace e_i with $ab_i?nondet() : e_i$. Unfortunately, this approach suffers from two problems. First, it allows modifying any expression, which results in extremely large search space. Secondly, it would return many absurd false diagnoses. For example, suppose that there is a loop, the loop body contains an assignment to the output variable, and there is no assignment to the output variable after the loop. Then this assignment would be returned as a diagnosis. A concrete example is the POWER program where the assignment $p = p * 2$ would be returned as a diagnosis. This is because $p = p * 2$ is replaced with $p = ab_i?nondet() : p * 2$, and in the last execution of the loop, we can always assign to p the correct value. By only allowing predicate switching, these two problems are avoided in our approach.

Conclusions

In this paper, based on Zhang *et al.*'s work on debugging via predicate switching, we proposed the bounded debugging via multiple predicate switching algorithm, which can be implemented by resorting to a SAT solver. We formalized the concepts of critical predicate sets and predicate-cut diagnoses. We proved that for conditional programs, BMPS is quasi-complete for RHS diagnoses; and for iterative programs, when the bound is sufficiently large, BMPS is also quasi-complete for RHS diagnoses. We analyzed the TCAS task of the Siemens Suite, and identified that 38 of the 41 faulty versions are predicate-cut faults. Initial experimentation with debugging small C programs showed that our approach is promising.

What distinguishes our work from existing ones is that it comes with a theoretical analysis. Secondly, inheriting from Zhang *et al.*'s approach, the search space for our approach is much reduced compared to those approaches based on arbitrary state changes. Thirdly, by only allowing predicate switching, our approach can avoid false diagnoses suffered by some existing ones.

For the future, we would like to extend the BMPS algorithm to accommodate procedures. Moreover, we would like to develop a full implementation of our algorithm and do a thorough experimentation and evaluation with it. The research methodology we used in this paper is to do theoretical abstraction of a practical debugging approach. We would like to continue with this methodology and perform theoretical analysis of other practical debugging approaches.

References

Clarke, E. M.; Kroening, D.; and Lerda, F. 2004. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

Griesmayer, A.; Staber, S.; and Bloem, R. 2006. Automated fault localization for C programs. In *Proc. First Workshop on Debugging and Verification*.

Groce, A. 2004. Error explanation with distance metrics. In *Proc. 10th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 108–122.

Levesque, H. J.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 1997. Golog: A logic programming language for dynamic domains. *J. Logic Programming* 31(1-3).

Liu, Y. 2008. A formalization of program debugging in the situation calculus. In *Proc. AAAI-08*.

Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.

Renieris, M., and Reiss, S. P. 2003. Fault localization with nearest neighbor queries. In *Proc. 18th IEEE Int. Conf. on Automated Software Engineering*, 30–39.

Rothermel, G., and Harrold, M. J. 1999. Empirical studies of a safe regression test selection technique. *Software Engineering* 24(6):401–419.

Zhang, X.; Gupta, N.; and Gupta, R. 2006. Locating faults through automated predicate switching. In *Proc. 28th Int. Conf. on Software Engineering*, 272–281.