A Formalization of Program Debugging in the Situation Calculus

Yongmei Liu
Department of Computer Science
Sun Yat-sen University
Guangzhou 510275, China
ymliu@mail.sysu.edu.cn

Abstract

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; it has also attracted attention from the AI community. However, existing approaches are mostly experiential; moreover, those model-based approaches are based on abstract models of programs, which lends an experiential flavor to the approaches, due to the heuristic nature of choosing an abstract model. We believe that it is necessary to establish a precise theoretical foundation for debugging from first principles. In this paper, we present a first step towards this foundation: using Reiter's theoretical framework of model-based diagnosis, we give a clean formalization of the program debugging task in the situation calculus, a logical language suitable for describing dynamic worlds. Examples are given to illustrate our formalization.

Introduction

Program debugging is one of the most time-consuming parts of the software development cycle. It generally refers to the task of locating and correcting faults in a program after they are detected by testing or model checking. In recent years, automatic debugging has been an active research area in software engineering (Renieris and Reiss 2003; Ball, Naik, and Rajamani 2003; Groce 2004; Jones 2004; Cleve and Zeller 2005; Zhang, Gupta, and Gupta 2006). Existing approaches are mostly experiential, *i.e.*, they depend on expert experience and heuristic information. For example, a common aspect of many approaches is to compare the statements occurring in correct runs and those occurring in failing runs. However, there is no guarantee for correctness or minimality of the diagnosis. How good the approaches are can only be evaluated empirically.

On the other hand, model-based diagnosis is a well-known AI technique for identifying faults in physical systems. There is an elegant theoretical foundation established by Reiter (1987). His theory requires only that the system can be described in a suitable logic, and the notion of diagnosis is defined through satisfiability in the logic. In the past decade, researchers have applied model-based diagnosis to program debugging (Wieland 2001; Wotawa 2002; Mayer and Stumptner 2007). They make use of abstract models of programs, due to the computational infeasibility of employing complete models. However, the choice of abstract models is of heuristic nature, which lends an experiential flavor to the approaches.

Despite the importance of experiential approaches, we believe that it is necessary to establish a precise theoretical foundation for debugging from first principles. The first ingredient of such a foundation would be a formal definition of the debugging task; the second ingredient would be an exploration of the associated computational problem including approximation techniques when necessary. In this paper, we present a first step towards this foundation: using Reiter's theoretical framework of model-based diagnosis, we give a formalization of program debugging in the situation calculus (Reiter 2001), a logical language suitable for describing dynamic worlds, and hence the dynamics of programs. In fact, a programming language Golog (Levesque et al. 1997) has been designed for the purpose of high-level robotic control and its formal semantics is defined in the situation calculus. The basic statements of Golog are primitive actions an agent can perform in the world. Algol-like programs can be treated as Golog programs, and hence we can obtain semantic definition of Algol-like languages in the situation calculus.

Intuitively, given a program P, a set of test cases including some failing ones, a diagnosis of P is a minimal set C of components of P such that a modification of P where components in C are replaced by possible substitutes, behaves correctly on all the test cases. Such a modification of P is called a repair of P. The debugging task is to return the set of all diagnoses. In this paper, we restrict our attention to Algol-like programs without procedures, which we call *while* programs. To formalize the above intuitive definition of debugging, we have to settle on two issues: (1) what are the program components to consider; and (2) what are the allowed substitutes for these components.

In this paper, we make the reasonable assumption that the control structure of the program is correct but errors may occur in assignments and control conditions of *if* and *while* statements. Note that each *while* program can be written into an equivalent one such that the control condition of each *if* and *while* statement is a uniquely named Boolean variable, by adding assignments to these variables. Also, note that each assignment can be rewritten into a finite sequence of assignments whose right hand sides involve at most one arithmetic, relational, or logical operator; we call such an assignment basic. Thus, each *while* program can be put into a normal form such that each control condition is a uniquely named Boolean variable and each assignment is basic. Now to answer (1), for normal form programs, it suffices to consider those components that are basic assignments. To answer (2), there are two possibilities: (a) the model of basic repair where a basic assignment is replaced by another basic assignment; and (b) the model of general repair where a basic assignment is replaced by a finite sequence of basic assignments. The model of general repair subsumes repair by deletion or addition of statements.

In the next section, we review the background work of this paper, that is, Reiter's theory of model-based diagnosis, existing research on program debugging, the situation calculus and Golog. Then we present our formalization of basic repair and extend it to a formalization of general repair. Finally, we conclude the paper.

Background Work

Reiter's Theory of Model-Based Diagnosis

Reiter (1987) distinguished between two kinds of approaches for identifying faults in physical systems: *experiential* ones which depend on expert experience and heuristic information, and *model-based* ones which are also called diagnosis from first principles. Roughly, the task of model-based diagnosis is this: given a model, *i.e.*, a description of the correct behavior of a system, and an observation of the system's behavior, determine those components which, when assumed to be functioning abnormally, will explain the discrepancy between the observed and correct behavior. Reiter (1987) developed a general theory of model-based diagnosis. Here we briefly review the basics of his theory.

A *system* to be diagnosed is given by a triple $(\mathrm{SD}, \mathrm{COMP}, \mathrm{OBS})$ where SD, the system description, is a set of logical sentences; COMP, the system components, is a finite set of constants; and OBS, an observation, is a finite set of logical sentences. There is a special predicate $\mathrm{AB}(c)$ which says that component c is abnormal.

Definition 1 A *diagnosis* for $(\mathrm{SD}, \mathrm{COMP}, \mathrm{OBS})$ is a minimal set $\Delta \subseteq \mathrm{COMP}$ s.t. the following is consistent:
$\mathrm{SD} \cup \mathrm{OBS} \cup \{\mathrm{AB}(c) \mid c \in \Delta\} \cup \{\neg \mathrm{AB}(c) \mid c \in \mathrm{COMP} - \Delta\}$.

The computational problem is then to determine the set of all diagnoses. There is a direct generate-and-test method, which, however, is too inefficient. Reiter proposed an algorithm, based on the following characterization of diagnoses through the concepts of conflict sets and hitting sets.

Definition 2 A *conflict set* for $(\mathrm{SD}, \mathrm{COMP}, \mathrm{OBS})$ is a set $\Delta \subseteq \mathrm{COMP}$ s.t. $\mathrm{SD} \cup \mathrm{OBS} \cup \{\neg \mathrm{AB}(c) \mid c \in \Delta\}$ is inconsistent.

Definition 3 Let C be a collection of sets. A *hitting set* for C is a set $H \subseteq \bigcup_{S \in C} S$ such that $H \cap S$ is non-empty for each $S \in C$. A hitting set is *minimal* if no proper subset of it is also a hitting set.

Theorem 1 *Δ is a diagnosis for $(SD, COMP, OBS)$ iff Δ is a minimal hitting set for the collection of its conflict sets.*

Reiter's algorithm computes all minimal hitting sets by constructing a hitting set tree (HS-tree) for all conflict sets, which are not explicitly given but are calculated as needed by an underlying theorem prover.

Program Debugging

Unlike the case for hardware diagnosis, existing approaches for program debugging are mostly experiential. There is no guarantee for correctness or minimality of the diagnosis. How good the debugging approaches are can only be evaluated empirically. A commonly used evaluation measure is the scoring function proposed by Renieris and Reiss (2003), based on the concept of *program dependency graph*. An evaluation benchmark suite 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. A well-known technique that has been used for debugging is *program slicing* (Weiser 1984), which is to extract statements relevant to the values of a given set of variables at some point of interest in the program. A large class of experiential debugging approaches are based on comparing correct and failing runs of the programs (Renieris and Reiss 2003; Ball, Naik, and Rajamani 2003; Groce 2004; Jones 2004; Cleve and Zeller 2005; Zhang, Gupta, and Gupta 2006).

In the past decade, researchers have applied model-based diagnosis to program debugging. The idea is to exchange the roles of the model and observation: the model now describes the behavior of the faulty program, while the observation specifies the behavior of the correct program. Mayer and Stumptner (2006) gave a survey of model-based debugging. They presented a general model of debugging as follows:

Definition 4 Let P be a program and T a set of test cases. Let COMP be the set of all the expressions in P, *i.e.*, the right hand sides of assignments, and the conditions in *if* and *while* statements. Let $\Delta \subseteq \mathrm{COMP}$, and let $P(\Delta)$ denote the program obtained from P by replacing each $e \in \Delta$ with $nondet()$, which returns a non-deterministic value. Δ is a diagnosis if for each test case t, there exists an execution of $P(\Delta)$ which satisfies t.

Obviously, this model is not computable in general. As a result, when model-based diagnosis techniques are applied to program debugging, various abstract models of programs are used. Mayer and Stumptner (2006) reviewed a dozen of abstract models, including models based on the control and data dependency between statements in a program (Wotawa 2002), and models based on abstract interpretation of programs, which concerns sound approximations of semantics of programs (Mayer and Stumptner 2007). However, how to choose an appropriate abstract model depends on expert experience and heuristic information, which lends an experiential flavor to the approaches. In fact, one of the two future challenges they identified for model-based debugging is how to select appropriate abstract models. Recently, Griesmayer *et al.* (2006) tackled the computational problem of the above general model from another perspective: they considered bounded-depth unfolding of while loops.

Apart from being informal, three drawbacks of the above general model (and also the work of Griesmayer *et al.*) are as follows. First, it does not capture our intuitive understanding of program debugging as presented in the introduction. Secondly, it only handles one fault type, *i.e.*, errors in expressions of programs; it cannot handle other fault types, for example, errors in left hand sides of assignments, and program repair by addition of statements. Thirdly, this model would result in many *false diagnoses*. For example, any function would get an absurd false diagnosis which consists only of the return statement, since for any test case t, there exists an execution of `return nondet()` which satisfies t. In fact, the other future challenge identified by Mayer and Stumptner is how to avoid false diagnoses.

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 use $\phi[s]$ to denote the formula obtained from ϕ by restoring s as the situation arguments to all fluents mentioned by ϕ.

Any domain of application is axiomatized by a basic action theory $\mathcal{D}$ with the following components:

1. The foundational axioms for situations;
2. Action precondition axioms;
3. Successor state axioms, one for each fluent;
4. Unique names axioms for the primitive actions;
5. An initial database, namely a set of axioms describing S_0.

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:
   $Do(\delta_1 \mid \delta_2, s, s') \overset{def}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$.
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 above 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:

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

A Formalization of Basic Repair

In this section, we present a formalization of basic repair in the situation calculus. We begin with the formal definition of *while* programs and a normal form for *while* programs. Then we present the situation calculus language of arithmetic programming. Next, we use Reiter's framework of model-based diagnosis to define the concepts of diagnosis and debugging. Finally, we give two examples to illustrate our formalization.

While Programs

In this paper, we restrict our attention to the debugging of *while* programs. 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.*, $L = L_P \cup \{-, /, \%, <, <=\}$. For each natural number $n > 0$, we simply use n to denote the term s^n0. We call terms of L *expressions*, quantifier-free formulas (*i.e.*, Boolean expressions) 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 expression t, $x := t \in \mathcal{W}$;
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** $\in \mathcal{W}$.

Normal Form Programs

We now define a normal form for *while* programs. Here we adopt the following convention of the C programming language: an assignment can be of the form $x := e$ where e is a condition, and x gets the value of 1 (resp. 0) when e is evaluated to be true (resp. false); a variable x can be used as a condition which is evaluated to be true (resp. false) when the value of x is non-zero (resp. zero). An assignment is basic if the right hand side (RHS) of it is n for some $n \geq 0$, or if its RHS involves only variables and at most one operator. We say that $S \in \mathcal{W}$ is in normal form if the following two conditions hold: 1. each assignment is basic; 2. each control condition is a uniquely named variable.

As mentioned in the introduction, in this paper, we make the reasonable assumption that the control structure of programs is correct. Thus when we debug normal form programs, it suffices to consider the repair of basic assignments.

Every *while* program can be converted into an equivalent program in normal form as follows. First, for each *while* statement **while** e **do** δ **od**, introduce a new variable x and convert the statement into $x := e$; **while** x **do** δ; $x := e$; **od**; and do similarly for each *if* statement. Then rewrite each assignment into a sequence of basic assignments by introducing new variables.

Example 1 Consider the following program PRIME which checks if the input n is a prime number:

```
prime := true;  i := 2;
while (prime and i < n/2) do
  if (n%i = 0) then prime := false; fi
  i := i + 1; od
```

It can be converted into PRIME$'$ in normal form:

```
prime := 1;  i := 2;  n1 := n / 2;
x1 := i < n1;  x2 := prime and x1;
while x2 do
  n2 := n % i;  x3 := n2 = 0;
  if x3 then prime := 0; fi
  i := i + 1;
  x1 := i < n1;  x2 := prime and x1; od
```

The Situation Calculus Language of Arithmetic Programming

We now present the situation calculus language of arithmetic programming $\mathcal{L}_{ap}$, which we will use to formalize the program debugging task. $\mathcal{L}_{ap}$ contains the following symbols:

- An infinite set of memory location constants $L_1, L_2, \ldots$
- An infinite set of test case constants $T_1, T_2, \ldots$
- An infinite set of component constants $O_1, O_2, \ldots$
- Operator constants OP $= \{$Val, Id, Succ, Add, Minus, Multi, Div, Mod, Less, Equal, LessOrEq, Not, And, Or$\}$
- A functional fluent $val(x, s)$ which denotes the value of memory location x in situation s
- Functions $init(t)$ and $final(t)$ which map test cases to their initial and final situations, respectively
- A predicate AB(c) which means component c is abnormal
- Action $assn(z, o, x, y)$, which assigns to memory location z value x if o is Val, or otherwise, the result of applying operator $o \in$ OP to the values of memory locations x and y.

Clearly, each normal form program can be expressed as a Golog program whose primitive actions are $assn(z, o, x, y)$.

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

1. The foundational axioms for situations.
2. Action precondition axioms:
$$Poss(assn(z,o,x,y),s) \equiv [o = \text{Minus} \supset val(x,s) \geq val(y,s)] \wedge [o = \text{Div} \vee o = \text{Mod} \supset val(y,s) \neq 0].$$
3. Successor state axioms:
$$val(z, do(a,s)) = v \equiv val(z,s) = v \wedge \neg(\exists oxy) a = assn(z,o,x,y) \vee (\exists oxy)[a = assn(z,o,x,y) \wedge \phi],$$
where ϕ is a disjunctive formula, with one disjunct for each operator. Here we only present the disjuncts for some operators; the others are similar.
$$\begin{aligned}\phi = \{&o = \text{Val} \wedge v = x \vee\\ &o = \text{Id} \wedge v = val(x,s) \vee\\ &o = \text{Succ} \wedge v = val(x,s) + 1 \vee\\ &o = \text{Add} \wedge v = val(x,s) + val(y,s) \vee\\ &o = \text{Less} \wedge [v = 1 \wedge val(x,s) < val(y,s) \vee\\ &\qquad v = 0 \wedge val(x,s) \geq val(y,s)] \vee\\ &o = \text{And} \wedge [v = 1 \wedge val(x,s) \cdot val(y,s) > 0 \vee\\ &\qquad v = 0 \wedge val(x,s) \cdot val(y,s) = 0] \vee\\ &\ldots\}\end{aligned}$$
This axiom says 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 memory locations.
4. Unique names axioms for $assn$: $\forall \vec{x}\vec{y}.assn(\vec{x}) = assn(\vec{y}) \supset \vec{x} = \vec{y}$.
5. Initial database:
   (a) Unique name axioms for the constants, *i.e.*, $d_1 \neq d_2$, for any two distinct constants d_1 and d_2
   (b) Domain closure axiom for actions, *i.e.*, $\forall a \exists \vec{x}.a = assn(\vec{x})$
   (c) The second-order axiomatization of Peano arithmetic
   (d) Definition of the symbols in $\{-, /, \%, <, <=\}$, *e.g.*,
   $$\forall xy[x < y \equiv \exists z(z \neq 0 \wedge x + z = y)]$$
   $$\forall xyz\{y > 0 \supset [x\%y = z \equiv z \geq 0 \wedge z < y \wedge \exists u(x = y \cdot u + z)]\}$$
   We do not have any constraints on the values of the memory locations in the initial situation S_0, and hence these values are left undetermined.

Diagnosis and Debugging

As mentioned in the introduction, for normal form programs, it suffices to consider the repair of basic assignments, and the model of basic repair is to replace a basic assignment with another one. We now formalize the notion of diagnosis via basic repair using Reiter's framework of model-based diagnosis. We begin with a formal definition of test cases.

Definition 5 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.

For example, $\langle n = 10, prime = 0\rangle$ is a test case for the program PRIME$'$.

A debugging problem is given by a tuple (P, COMP, CT, FT) where P is a normal form program written in Golog, COMP $= \{O_1, \ldots, O_n\}$ where each O_i is the identifier for an $assn$ action α_i in P, CT is a set of test cases where P behaves correctly, and FT is a non-empty set of failing test cases for P. To use Reiter's framework of model-based diagnosis, we first define the system description SD and the observation OBS as follows.

We introduce n action variables $a_1, \ldots, a_n$, and let P' denote P with each α_i replaced by a_i. We let SD(P, COMP)

denote the sentence

$$\exists\vec{a}\{(\forall t)Do(P', init(t), final(t))\wedge \bigwedge_{i=1}^{n}[\neg\text{AB}(O_i)\supset a_i=\alpha_i]\},$$

which says that there exists a substitution of the abnormal components α_i's by a_i's such that the execution of the resulting program makes the situation transfer from $init(t)$ to $final(t)$, for each test case t.

Let ϕ be an assertion. We use $\phi@s$ to denote the situation calculus formula obtained from ϕ by replacing each variable x with $val(x,s)$, *e.g.*, $(n=10)@s$ is $val(n,s)=10$. Given a set Γ of test cases, we let OBS(Γ) denote the sentence

$$\bigwedge_{T\in\Gamma}[In_T@init(T)\supset Out_T@final(T)],$$

which says that for each test case T, if In_T holds in $init(T)$, then Out_T holds in $final(T)$.

Now adapting Definition 1, we have

Definition 6 A *diagnosis* for (P,COMP,CT,FT) is a minimal set $\Delta\subseteq$ COMP s.t. the following is consistent wrt $\mathcal{D}_{ap}$:

$$\text{SD}(P,\text{COMP})\cup\text{OBS}(CT\cup FT)\cup$$
$$\{\text{AB}(c)\mid c\in\Delta\}\cup\{\neg\text{AB}(c)\mid c\in\text{COMP}-\Delta\}.$$

Similarly, we can adapt Definition 2, and define the concept of *conflict set*. As a result, Theorem 1 also holds here, *i.e.*, Δ is a diagnosis iff Δ is a minimal hitting set for the collection of conflict sets. Therefore, under the assumption that whether a set is a conflict set is decidable, Reiter's hitting set algorithm could be used to compute the set of all diagnoses.

Of course, since $\mathcal{D}_{ap}$ involves arithmetic, whether a set is a diagnosis or conflict set is undecidable in general, and hence it is necessary to explore approximation algorithms, for which we define the concepts of soundness and completeness as follows.

Definition 7 A collection C of nonempty subsets of a set A is called an *antichain* in A if no set in C is properly contained in another set from C.

For example, let $A=\{1,2,3,4\}$, and $C=\{\{1,2\},\{2,3\},\{1,3,4\}\}$. Then C is an antichain in A. Clearly, the set of all diagnoses is an antichain. Our basic requirement for an approximate debugging algorithm is to return an antichain.

Definition 8 Let C and C' be two antichains in A. We say that C subsumes C', written $C\le C'$, if for any $S\in C$, there exists $S'\in C'$ such that $S\subseteq S'$.

It is easy to show that $\le$ is a partial order.

Definition 9 Given (P,COMP,CT,FT), we say that a debugging algorithm is *sound* (resp. *complete*) if it returns an antichain in COMP which subsumes (resp. is subsumed by) the set of all diagnoses.

The intuition here is that any diagnosis returned by a sound debugging algorithm should be a subset of a real diagnosis, and a complete debugging algorithm should have the property that any real diagnosis is a subset of a diagnosis returned by the algorithm. A trivial sound debugging algorithm is to return the empty set, and a trivial complete debugging algorithm is to return the singleton of COMP.

We now give two examples to illustrate the above formalization.

Example 2 We have a very simple program:

```
1. S := A and B;
2. D := not S;
3. E := S or C;
```

There is a correct test case
$T_1=\langle A=1\wedge B=0\wedge C=0, D=1\wedge E=0\rangle$,
and a failing test case
$T_2=\langle A=1\wedge B=1\wedge C=0, D=0\wedge E=0\rangle$.

Then our SD is:

$$(\exists a_1,a_2,a_3)\{(\forall t)Do(a_1;a_2;a_3,init(t),final(t))\wedge$$
$$[\neg\text{AB}(O_1)\supset a_1=assn(S,\text{And},A,B)]\wedge$$
$$[\neg\text{AB}(O_2)\supset a_2=assn(D,\text{Not},S,\_)]\wedge$$
$$[\neg\text{AB}(O_3)\supset a_3=assn(E,\text{Or},S,C)]\},$$

and our OBS is:

$$[val(A,init(T_1))=1\wedge val(B,init(T_1))=0\wedge$$
$$val(C,init(T_1))=0\supset$$
$$val(D,final(T_1))=1\wedge val(E,final(T_1))=0]\wedge$$
$$[val(A,init(T_2))=1\wedge val(B,init(T_2))=1\wedge$$
$$val(C,init(T_2))=0\supset$$
$$val(D,final(T_2))=0\wedge val(E,final(T_2))=0]$$

We have two conflict sets: $\{O_1,O_3\}$ and $\{O_2,O_3\}$. To see why $\{O_1,O_3\}$ is a conflict set, assume that O_1 and O_3 are normal. For the program to behave correctly on T_2, since the output value of D is 0, the second statement must be an assignment to D. Then when O_3 is executed, S has the value 1, and hence the output value of E is 1, a contradiction. By Theorem 1, we have two diagnoses: $\{O_1,O_2\}$ and $\{O_3\}$.

Example 3 Consider the following program FACTORIAL which intends to compute the factorial of the input n:

```
1. fac := 1; 2. i := 1; 3. b := i < n;
4. while b do
5.    fac := fac * i; 6. i := i + 1;
7.    b := i < n; od
```

Suppose it has two failing test cases $\langle n=2, fac=1\rangle$ and $\langle n=6, fac=120\rangle$. Then $\{O_7\}$ is a diagnosis, since we can replace it with `b:=i<=n` and get the correct behavior.

A Formalization of General Repair

In this section, we extend the formalization of basic repair to a formalization of general repair.

General Repair

The model of basic repair has its limitations, for example, it cannot handle repair by addition of statements. The model of general repair is to replace a basic assignment with a finite sequence of basic assignments. To formalize this model, we note the relationship between situations and finite sequences of actions: a situation is resulted from executing a finite sequence of actions in the initial situation. Let s be a situation. We introduce the notation $seq(s)$ to denote the sequence of actions contained in s, and we define its semantics through an abbreviation $Do(seq(s),s_1,s_2)$, which intuitively means that the execution of the actions contained in s makes the situation transfer from s_1 to s_2. To give the formal definition, we have to resort to second-order logic. We add the following sentence to the basic action theory $\mathcal{D}_{ap}$:

$$Do(seq(s), s_1, s_2) \overset{def}{=} (\forall R).\{(\forall s)R(S_0, s, s) \wedge (\forall a, s, s_1, s_2)[R(s, s_1, s_2) \supset R(do(a, s), s_1, do(a, s_2))]\} \supset R(s, s_1, s_2).$$

We introduce n situation variables $s_1, \ldots, s_n$, and use P^* to denote program P with each α_i replaced by $seq(s_i)$. Now our SD is as follows:

$$\exists \vec{s}\{(\forall t)Do(P^*, init(t), final(t)) \wedge \bigwedge_{i=1}^{n}[\neg \mathrm{AB}(O_i) \supset s_i = do(\alpha_i, S_0)]\},$$

Intuitively, if O_i is abnormal, we replace α_i with the sequence of actions contained in s_i.

Bounded-Length Repair

In practice, a model of repair that lies between basic repair and general repair would suffice. This is the model of bounded-length repair where a basic assignment can be replaced by a bounded-length sequence of basic assignments. Its formalization can be easily obtained from the formalization of general repair by adding constraints. Let $k \geq 0$. We introduce an abbreviation $length(s) \leq k$ as follows. Intuitively, it means that s contains at most k actions.

1. $length(s) \leq 0 \overset{def}{=} s = S_0$;
2. $length(s) \leq k + 1 \overset{def}{=} length(s) \leq k \vee (\exists a, s')[lenghth(s') \leq k \wedge s = do(a, s')]$.

Now our SD is this:

$$\exists \vec{s}\{(\forall t)Do(P^*, init(t), final(t)) \wedge \bigwedge_{i=1}^{n} length(s_i) \leq k \wedge [\neg \mathrm{AB}(O_i) \supset s_i = do(\alpha_i, S_0)]\}.$$

Example 4 Consider the following program which intends to compute the nth Fibonacci number:

```
1. x := 1;         2. y := 1;
3. i := 2;         4. b := i < n;
5. while b do
6.   x := y;       7. y := x + y;
8.   i := i + 1;   9. b := i < n; od
```

We have a correct test case $T_1 = \langle n = 3, y = 2 \rangle$ and two failing test cases $T_2 = \langle n = 4, y = 4 \rangle$ and $T_3 = \langle n = 5, y = 8 \rangle$. If we consider the model of bounded-length repair where the bound is 2, then neither $\{O_6\}$ nor $\{O_7\}$ is a diagnosis, but $\{O_6, O_7\}$ is a diagnosis, since we can get the correct behavior by replacing O_6 with `t:=x; x:=y;` and replacing O_7 with `y:=x+t;`.

Conclusions

Program debugging is one of the most time-consuming parts of the software development cycle. Existing approaches are mostly experiential; those model-based approaches are based on abstract models of programs, which adds an experiential flavor to the approaches. We believe that it is necessary to establish a precise theoretical foundation for debugging from first principles. To the best of our knowledge, this paper presented the first clean logical description of the program debugging task. We focused our attention on programs without procedures, considered both the basic and general models of repair, and defined the concepts of soundness and completeness for approximate debugging methods. Our formalization has the advantages of being conceptually simple, handling multiple test cases, supporting general fault types, and avoiding false diagnoses suffered by existing models. Such a formalization can serve as the basis for further research on debugging from first principles. In the future, we would like to extend our formalization to include procedures. Most importantly, based on our formalization, we would like to explore the computational problem including approximation techniques that are sound or complete.

Acknowledgments

I am grateful to Fangzhen Lin for many helpful discussions about this work. I thank Yilan Gu and Ron Petrick for their help with this paper. Thanks also to the anonymous reviewers for useful comments.

References

Ball, T.; Naik, M.; and Rajamani, S. K. 2003. From symptom to cause: localizing errors in counterexample traces. In *Proc. 30th Symp. Principles of Programming Languages*.

Cleve, H., and Zeller, A. 2005. Locating causes of program failures. In *Proc. 27th Int. Conf. on Software Engineering*, 342–351.

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.

Jones, J. A. 2004. Fault localization using visualization of test information. In *Proc. 26th Int. Conf. on Software Engineering*.

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).

Mayer, W., and Stumptner, M. 2006. Model-based debugging - state of the art and future challenges. In *Proc. First Workshop on Debugging and Verification*.

Mayer, W., and Stumptner, M. 2007. Abstract interpretation of programs for model-based debugging. In *Proc. IJCAI-07*.

Reiter, R. 1987. A theory of diagnosis from first principles. *Artificial Intelligence* 32(1):57–95.

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.

Weiser, M. 1984. Program slicing. *IEEE Trans. Software Eng.* 10(4):352–357.

Wieland, D. 2001. *Model-Based Debugging of Java Programs Using Dependencies*. Ph.D. Dissertation, TU Wien.

Wotawa, F. 2002. On the relationship between model-based debugging and program slicing. *Artificial Intelligence* 135:124–143.

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