Debugging Functional Programs

Markus Stumptner and Franz Wotawa
Technische Universität Wien
Institut für Informationssysteme
Paniglgasse 16, A-1040 Wien, Austria,
Email: {mst,wotawa}@dbai.tuwien.ac.at

Abstract

In this paper, we use a logic-based system description for a simple (non-logic) functional language to examine the ways in which a diagnosis system can use its system description to improve debugging performance. The key concepts for this are that the notion of expression replacement, which is the basis for repairing a program, can also serve as a fundamental heuristic for searching the source of an error. We formally define replacements in terms of fault modes, explicitly define a replacement order, and present an algorithm that uses the replacement heuristic for finding diagnoses. Finally, we incorporate the use of multiple test cases and discuss the resulting reduction of the number of computed diagnoses.

Introduction

Although model-based diagnosis (MBD) (Rei87; dKW87) provides a general diagnosis framework, most of its applications are within the hardware domain or use models of the physical world. However, several attempts have been made to extend the application of diagnosis methods to software debugging. A model for diagnosing logic programs was introduced in (CFD93) and further studied in (BP94; Bon94; Bon96). In (FSW96), the model-based diagnosis of circuit designs written in the VHDL hardware specification language used an abstract model that allows diagnosing very large programs within reasonable time. In (SW96), we presented a graph-based model for diagnosing a functional language, EXP.

All these approaches have in common that they use only a model which can be directly derived from the given program and the used programming language. No additional formal specification must be available, so that the diagnosis system can be directly integrated into a traditional software development process. In this paper, we give a concise logic-based model for a variant of EXP which can be adapted to other functional, or sequential assignment languages, and has the advantage of being executable over the definition of (SW96). We keep the ability to introduce models for expression replacements that explain a faulty behavior, and discuss the application of multiple test cases, which can reduce the number of diagnoses by excluding certain replacements - a phenomenon that is dual to the concept of physical impossibility. This helps to focus on faulty program parts without using too sophisticated models. User interaction (the user as oracle) is replaced by sets of prespecified test values.

To begin, we briefly recapitulate the basic MBD definitions (see (Rei87; SD89; Wot96)). A diagnosis problem consists of a system description (SD) describing the behavior of components (COMP) and their interconnections. The component behavior is associated with component modes. For every component at least the correct behavior ($\neg ab$ mode) must be specified. Let OBS be a set of observations. A diagnosis for $(SD, COMP, OBS)$ is a set of mode assignments Δ, associating one mode with every component, such that $SD \cup OBS \cup \{m(C) | m(C) \in \Delta\}$ does not lead to a contradiction.

This paper is organized as follows. First, syntax and semantics of the language of expressions (EXP) is given, followed by the introduction of the associated system description. Afterwards, replacements and the use of multiple test cases are discussed. The final section concludes the paper and gives future research directions.

Modelling Programs for Diagnosis

Programs can be viewed as static and dynamic objects. The program's static part is limited by the syntax given by the used programming language, while the dynamic part is related to the semantics. Executing a program causes the evaluation of constants, variables, functions, etc. using *environments* (which map variables to their actual values and function identifiers to their definition or *body*) which can be changed during evaluation. In this paper, we introduce a model for a simple functional programming language EXP with the following syntax:

expression ::=
  `if` *expression* `then` *expression* `else` *expression* `end if`
  | `(` *expression* `)`
  | *expression* *bin_op* *expression*
  | *funct_id* `(` *expression_list* `)`

| *var_id*
| *const*

expression_list ::=
ϵ
| expression expression_list_rest

expression_list_rest ::=
ϵ
| `,` expression expression_list_rest

It is assumed that VS contains all variable symbols, FS all built-in functions, FV all user-defined functions, and A all constants. Built-in functions (including predicates) represent the basic functionality of the language. They include functions like + or * depending on the implemented data types. While constants evaluate to the same value every time, variables evaluate to values depending on the current state of the environment that stores the values. Likewise, the bodies of user-defined functions are stored in a function environment. Formally, we define a *variable environment* as a function $\iota : VS \mapsto VALUE$ and a *function environment* as a function $\delta : FV \mapsto \mathcal{EXP}$ associating expressions with function symbols. We denote the set of all variable environments by $VENV$, the set of all variable environments by $FENV$.

In addition to the function environment we introduce the function $variables$ which returns all variables used in the body of a user-defined function.

The semantics of EXP, given in a declarative manner, describe how EXP expressions are evaluated. We use a call-by value semantics for this paper. The semantics are given by an evaluation function $\Gamma : EXP \times VENV \times FENV \mapsto VALUES$ which maps expressions (programs) together with the given environments to a set of result values.

$V \in VS \Rightarrow \Gamma(V, \iota, \delta) = \iota(V)$
$C \in A \Rightarrow \Gamma(C, \iota, \delta) = C$
$\Gamma(\texttt{(}E\texttt{)}, \iota, \delta) = \Gamma(E, \iota, \delta)$
$F \in FS \Rightarrow$
$\Gamma(F(e_1, \ldots, e_n), \iota, \delta) = F(\Gamma(e_1, \iota, \delta), \ldots, \Gamma(e_n, \iota, \delta))$
$F \in FS \Rightarrow \Gamma(e_1 F e_2, \iota, \delta) = \Gamma(e_1, \iota, \delta) F \Gamma(e_2, \iota, \delta)$
$\Gamma(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \texttt{ end if}, \iota, \delta) =$
$$= \begin{cases} \Gamma(e_2, \iota, \delta) & \text{if } \Gamma(e_1, \iota, \delta) = true \\ \Gamma(e_3, \iota, \delta) & \text{otherwise} \end{cases}$$
$F \in FV \Rightarrow \Gamma(F(e_1, \ldots, e_n), \iota, \delta) = \Gamma(\delta(F), \iota', \delta)$
where $v_i \in variables(F) : \iota'(v_i) = \Gamma(e_i, \iota, \delta)$ and
$v \notin variables(F) : \iota'(v) = \iota(v)$.

We demonstrate the usage of the semantics on a small EXP example. Given the following program:

```
twodimln(X,Y)
  if (X > 0) and (Y > 0) then
    ln(X * Y)
  else
    0
  end if
```

the body of the user-defined function `twodimln` is given by

$$\delta(\texttt{twodimln}) = \begin{cases} \texttt{if (X > 0) and (Y > 0)} \\ \texttt{then ln(X * Y)} \\ \texttt{else 0 end if} \end{cases}$$

The variables used are $variables(twodimln) = \{X, Y\}$ which is a subset of VS. FS includes the functions (predicates) '`>`', '`and`', '`ln`', and '`*`'. The constant `0` is contained in A.

The following shows the beginning of the evaluation of the function:

$\Gamma(\texttt{twodimln(1,2)}, \iota, \delta)$
$\iota'(\texttt{X}) = \Gamma(1, \iota, \delta) = 1, \iota'(\texttt{Y}) = \Gamma(2, \iota, \delta) = 2$
$\Gamma(\delta(\texttt{twodimln}), \iota', \delta)$
$\Gamma(\texttt{if (X > 0) and (Y > 0) then}$
$\texttt{ln(X * Y) else 0 end if}, \iota', \delta)$
$\Gamma(\texttt{(X > 0) and (Y > 0)}, \iota', \delta)$
$and(\Gamma(\texttt{(X > 0)}, \iota', \delta), \Gamma(\texttt{(Y > 0)}, \iota', \delta))$
$> (\Gamma(\texttt{X}, \iota', \delta), \Gamma(\texttt{0}, \iota', \delta))$
$\iota'(\texttt{X}) = 1$
0
$true$
$> (\Gamma(\texttt{Y}, \iota', \delta), \Gamma(\texttt{0}, \iota', \delta))$
$\iota'(\texttt{Y}) = 2$
0
$true$
$true$
$\Gamma(\texttt{ln(X * Y)}, \iota', \delta)$
$ln(\Gamma(\texttt{X * Y}, \iota', \delta)$
$*(\Gamma(\texttt{X}, \iota', \delta), \Gamma(\texttt{Y}, \iota', \delta))$
$\iota'(\texttt{X}) = 1$
$\iota'(\texttt{Y}) = 2$
2
0.693
0.693
0.693
0.693

To use MBD for software debugging we must identify diagnosis components and build a system description. Because EXP allows building programs out of expressions, we use expressions as diagnosis components. To differentiate between separate occurrences of textually identical expressions we introduce a function κ which assigns an unique number to every expression. A system description for EXP must have the same properties as the semantics. Results obtained by the semantics must be also derivable by the system description.

The diagnosis components for our above example `twodimln` are given by:

C_0 = `if (X > 0) and (Y > 0) then ln(X*Y) else 0 end if`,
C_1 = `(X > 0) and (Y > 0)`,
C_2 = `(X > 0)`,
C_3 = `X`,
C_4 = `0`,
C_5 = `(Y > 0)`,
C_6 = `Y`,
C_7 = `0`,
C_8 = `ln(X*Y)`,
C_9 = `X*Y`,

C_{10} = X,
C_{11} = Y,
C_{12} = 0

If we examine the data flow between the components, we find that they form a tree-structured system with C_0 as top component. Such a system can be diagnosed very effectively (SW97). In the next step, we declare a system description EXPSD for EXP programs and prove its completeness and correctness related to EXP semantics.

The semantics definition in (SW96) turned out to be somewhat unwieldy. The formal definitions where complex, and required another step to be mapped to a logical representation for a diagnosis system. Therefore we use a PROLOG-like description, which also has the advantage of being executable. To handle unknown values, i.e., values that cannot be derived from the description and observations, we introduce the distinguished value ϵ. The evaluation is defined using a predicate

```
eval(Expr, VarEnv, Val, Step)
```

where `Expr` denotes the expression to be evaluated, `VarEnv` a list of variable value pairs, `Val` the value returned by the evaluation, and `Step` the number of iteration steps. The variable environment of our running example is expressed by `[[X,1],[Y,1]]`.

The function environment is represented by the predicates `body/2` and `variables/2`. To evaluate a call to `twodimln`, the facts `body(twodimln,if (X > 0) and (Y > 0) then ln(X*Y) else 0 end if)` and `variables(twodimln,[X,Y])` are added to the system description.

1. **Constants** are evaluated to themselves.

```
eval(Expr,VarEnv, Expr, Step) :- ¬ab(Expr).
```

2. **Variables** are evaluated to the associated value in the current variable environment.

```
eval(Expr,VarEnv,V, Step)) :-
  ¬ab(Expr), value(VarEnv,Expr,V).
```

where `value` is defined as follows:

```
value([[X,V]|Rest], X, V).
value([], X, ε).
value([[Y,V]|Rest], X, V') :-
  Y ≠ X, value(Rest, X, V').
```

3. **Built-in functions**

The following rule belongs to the system description for every built-in function (or predicate) used in the program (e.g., '+', or '-').

```
eval(F(E1,...,En),VarEnv,V,Step)) :-
  ¬ab(F(E1,...,En)),
  eval(E1,VarEnv,V1,Step),
  ...
  eval(En,VarEnv,Vn,Step),
  F(V1,...,Vn,V).
```

The behavior of the function is given by the $\mathtt{F}/n+1$ predicate. As an example, for the logical `and` function the behavior could be described by:

```
and(true,true,true).  and(false,X,false).
and(X,false,false).  and(true,ε,ε).
and(ε,true,ε).
```

4. **Conditional expression** `E = if E1 then E2 else E3 end if`

```
eval(E,VarEnv,V,Step)) :-
  ¬ab(E), eval(E1,VarEnv,true,Step),
  eval(E2,VarEnv,V,Step).

eval(E,VarEnv,V,Step)) :-
  ¬ab(E), eval(E1,VarEnv,true,Step),
  eval(E3,VarEnv,V,Step).

eval(E,VarEnv,ε,Step)) :-
  ¬ab(E), eval(E1,VarEnv,ε,Step).
```

5. **User-defined functions** are evaluated by changing the environment and executing the body using it.

```
eval(F(E1,...,En),VarEnv,V,Step)) :-
  body(F,B), variables(F,Vars),
  ¬maximumSteps(Step),
  changeEnv(VarEnv,[E1,...,En],Vars,VE,
      Step),
  eval(B,VE,V,Step+1).

changeEnv(VarEnv,[],[],[],Step).
changeEnv(VarEnv,[E|Rest],[Var|VRest],
     [[Var,V]|ERest],Step) :-
  eval(E,VarEnv,V,Step),
  changeEnv(VarEnv,Rest,VRest,ERest,Step).

maximumSteps(Step) :- Step >= maxStep.
```

6. **Undefined Behavior**

If a component is assumed to behave incorrectly, the undefined value is returned as result.

```
eval(E,VarEnv,ε,Step) :- ab(E).
```

It can be shown that the above system description produces an evaluation that corresponds exactly to the semantics of EXP.

To complete the system description for model-based diagnosis we must introduce the concept of observations.

Therefore we introduce the predicate `observe/3` using the expression, the environment, and the expected value as arguments. An observation for our example can be `observe(twodimln(1,1),Env,0)`.

In this case no environment ist specified because the function call does not make use of variables. We say that an observation contradicts the system description if the value derived from SD is not equal to the observation. This rule is added to our system description.

```
contradiction :-
  eval(E,Env,V,Step), observe(E,Env1,V1),
  equalEnv(Env,Env1), V ≠ ε, V1 ≠ V.
```

Observations are facts and cannot contain the undefined value ϵ. The predicate `equalEnv/2` is true if both environments contain the same elements.

With the system description EXPSD we can compute (minimal) diagnoses as described by Reiter (Rei87). For example assume that instead of `(Y > 0)` the programmer has written `(Y > 1)` as subcondition. Using the test case `twodimln(2,1)` we can compute the following 6 single diagnoses:

$\{C_0\}$ = $\{$`if (X > 0) and (Y > 1) then ln(X*Y) else 0 end if`$\}$,
$\{C_1\}$ = $\{$`(X > 0) and (Y > 1)`$\}$, $\{C_5\}$ = $\{$`(Y > 1)`$\}$,
$\{C_6\}$ = $\{$`Y`$\}$, $\{C_7\}$ = $\{$`1`$\}$, $\{C_{12}\}$ = $\{$`0`$\}$

Among 13 diagnosis components, the diagnosis engine has found 6 diagnosis candidates. In the rest of the paper, we discuss how to further reduce the number of suspected components. 1 First, by using replacements to attempt to correct a program, as described in (SW96). The second idea is to use multiple test cases and the information related to them. Finally, we can introduce fault modes and their corresponding behavior. We can easily prove whether a condition within a conditional statement is a single bug or not by inverting the conditional statement's behavior. Unfortunately, applying this fault mode has a significant drawback. If we are diagnosing a recursive function, inverting the condition may lead to an infinite loop. In the example we can use this mode effectively - if the condition is be inverted, correct behavior will result. Therefore, only the diagnoses $\{C_1\}$, $\{C_5\}$, $\{C_6\}$ and $\{C_7\}$ remain.

Software Debugging as Search for Replacements

As argued in (CFD93), applying model-based diagnosis to software debugging reduces the number of diagnosis candidates compared to classical program debugging techniques such as (Sha83), and this is already the case with a straightforward application of the standard diagnosis approach to debugging. On the other hand, we are normally not only interested in finding an expression explaining the wrong behavior, we also want to give a correction that will produce the expected behavior instead. In the hardware domain this correction is simple. Only the defective component of one type must be replaced by a working component of the type specified in the hardware design[1]. In this case, the search space for replacement is of size 1. In the software domain (or design domain), a replacement can be every buildable function, and in this case, the search space is infinite. However, attempting to correct a program also has the benefit that diagnoses that cannot be combined with a meaningful correction of the error can be excluded from consideration. This was the main reason for the introduction of expression replacements in (SW96). Every replacement represents a possible correction for the program, and to handle the infinite search space, we need to define an effective ordering on replacements. Note that even in simple cases, it is not always clear what criteria should be applied to produce this order. For example, is replacing the variable `X` by the constant `XC` a smaller change than replacing it by the variable `Y`?

We define replacements as expressions, i.e., a replacement is an element of EXP. We introduce an ordering on the replacements that are used instead of a given original expression. This ordering is specified by defining a function $replace$ associating a replacement of a given depth with an expression.

Variables A variable can be replaced by another variable, constant, and function.

$$replace(V,0) = \{V\}$$
$$replace(V,1) = VS \setminus \{V\}$$
$$replace(V,2) = A \cup \{f | f \in FS \cup FV, arity(f) = 0\}$$
$$replace(V,n) = \{e | e \in EXP, size(e) = n, n > 2\}$$

Constants Replacements for constants are similar to variable replacements.

$$replace(C,0) = \{C\}$$
$$replace(C,1) = A \setminus \{C\}$$
$$replace(C,2) = VS \cup \{f | f \in FS \cup FV, arity(f) = 0\}$$
$$replace(C,n) = \{e | e \in EXP, size(e) = n, n > 2\}$$

Others Function Calls, Predicates, and Conditionals can be replaced by any other expression.

$$\begin{aligned} & replace(F(e_1,\ldots,e_n),j) = \\ & \quad replaceSub(F(e_1,\ldots,e_n),j) \cup \\ & \quad replaceAdd(F(e_1,\ldots,e_n),j) \cup \\ & \quad replaceSel(F(e_1,\ldots,e_n),j) \end{aligned}$$

where the subfunctions are given by:

- Change the function and use a subset of the original arguments. The arguments are not modified.

[1]This formulation includes the possibility that a component of a wrong type has been used.

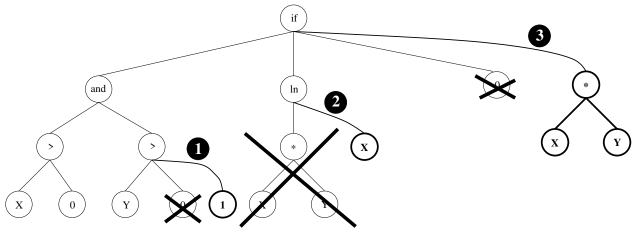

Figure 1: Three Ways of Applying Replacements

$$replaceSub(F(e_1,\ldots,e_n), k+(n-m)) = \Big\{G(f_1,\ldots,f_m) | G \in FS \cup FV, arity(G) = m \leq n, (f_1,\ldots,f_m) \text{is a subsequence of } (e_1,\ldots,e_n)\Big\}$$

where $k = \begin{cases} 1 & \text{if } F \neq G \\ 0 & \text{otherwise} \end{cases}$

- Change the function and add new arguments.
  $$replaceAdd(F(e_1,\ldots,e_n), k+(m-n)+\sum_{i=n\ldots m} size(e_i)) = \Big\{G(e_1,\ldots,e_n,e_{n+1}\ldots,e_m) | G \in FS \cup FV, arity(G) = m > n, \forall_{i=n\ldots m} e_i \in EXP\Big\}$$
  where $k = \begin{cases} 1 & \text{if } F \neq G \\ 0 & \text{otherwise} \end{cases}$
- Select an argument. The function is discarded and the argument is not changed.
  $$replaceSel(F(e_1,\ldots,e_n),n) = \{e | e \in \{e_1,\ldots,e_n\}\}$$

For simplicity, we do not allow permutation of arguments in the definitions given here and do not include any in our examples. Replacements for the subexpression `Y < 1` of our example are:

Order	Replacements
0	`Y < 1`
1	`Y <= 1, Y > 1, Y = 1,...`
2	`-Y, -1,...`
3	`min(Y,1,2),...`

Note that for diagnosis purposes we can ignore replacements of subexpressions, as the need for replacement of only a subexpression would be directly expressed by that subexpression becoming a diagnosis candidate of its own.

Figure 1 illustrates the application of replacements to the syntactical representation of a program as a tree. In case (1) the constant *0* is replaced by the constant *1*. (2) gives the case where an argument has been selected from a function. And (3) shows a replacement of a constant by a function with two arguments.

The set of possible replacements can be further reduced by considering type compatibility. E.g., a multiplication function should be only replaced by another function returning a number. The same holds for variables and constants.

Algorithm DEBUG

1. $n := 0$.
2. Compute diagnoses using replacements of order $\leq$ n.
3. If the diagnosis set is good enough, stop and return the current set.
4. Otherwise $n := n + 1$.
5. If the given limit is reached, stop the algorithm and return that no sufficient diagnosis has been found. Otherwise goto step 2.

Figure 2: Debugging using Replacements

To use the replacements in our system description we must add a fault mode `replace`. We represent this fault mode by the predicate `replace/2` that can be constructed from the computed replacements. Let E be an expression, and n be the order of the replacements. For every replacement used for diagnosis we have to add the following (ground) fact to the system description:

```
replace(E,R).
```

where $\mathtt{R} \in \bigcup_{i \leq n} replace(\mathtt{E}, i)$.

The behavior associated with the fault mode is specified in a natural way by:

```
eval(E,Env,V,Step) :-
  replace(E,R), eval(R,Env,V,Step).
```

where it is assumed that the expression `R` and all of its subexpressions behave correctly, i.e., $\neg$`ab`.

The DEBUG algorithm (from (SW96)) computes diagnoses using replacements (see figure 2). Diagnoses are searched for incrementally, starting with replacements of order 0 and stopping after reaching a specified limit. Note that only replacements of minimal order are taken into account, e.g., if a replacement of order i explains the fault, no replacements of order $j > i$ must be considered.

Assuming that consistency checks take linear time, the time complexity of diagnosis is of order $o(2^{|COMP|})$ where $COMP$ denotes the set of diagnosis components. Adding as set of fault modes $MODE$, e.g., as done by introducing replacements, increases time complexity to $o(|MODE|^{|COMP|})$. The time for computing single diagnoses is of order $o(|MODE| \cdot |COMP|)$, limiting the applicability of this approach to small programs. However, using focusing techniques and doing incremental debugging using DEBUG helps to increase practicability. Focusing can be done using a more abstract model or structural properties of the program, e.g., using only those expressions evaluated for a testcase as diagnosis components, or (static) program slicing (see (Wei84)). Another way is the use of multiple testcases as described in the next section.

Using Multiple Test Cases

In this section, we are going to explain how multiple test cases can help locating faults within a program. Before applying the results to the software domain we introduce the basic idea using a hardware design example. Figure 3 shows the circuit diagram of a ripple-carry adder built out of inverters and AND gates.

The values for each signal can be found in the figure. Now consider an incorrect version of the adder, where the AND gate *A1* has been replaced by an OR gate (although named *A1*).

In this case the output **Z** has the constant value **1** which is obviously not expected. In order to fix the fault we compute all (minimal) diagnoses using the standard model-based approach and obtain the following diagnoses.

X	Y	Diagnoses
0	0	$I1, A1, I3, A4, I5$
1	0	$\{\}$
0	1	$\{\}$
1	1	$A1, I3, A4, I5$

The result is negative in the sense that it does not eliminate a significant set of diagnoses. On the other hand, we can prove that some (single) diagnoses cannot actually correspond to the error because they cannot be replaced by a valid boolean function. For example, have a look at diagnosis $A4$. Assuming that all other components work correctly and only $A4$ fails, leads to the following input output behavior.

X	Y	I1(A4)	I2(A4)	O(A4)	Expected O(A4)
0	0	0	1	0	1
1	0	0	1	0	0
0	1	1	0	0	0
1	1	0	1	0	1

From the table we see that **A4** has to produce 1 and 0 for the same input vector which is of course impossible. Therefore, we can eliminate $A4$ from the set of diagnoses[2]. The same argument can be used to eliminate $I3$ and $I5$. Therefore, the single diagnosis $A1$ remains. Using the repair approach, only the function *or* allows establishing the expected behavior.

The same results as presented before can be obtained in the software domain. Consider for example the following functional program (wrongly) implementing the equality predicate:

[2] In the case where internal states are possible, this conclusion cannot be applied in general. In this case the internal state must be represented as an additional input, e.g., the sequential circuit must be converted to a combinatorial one.

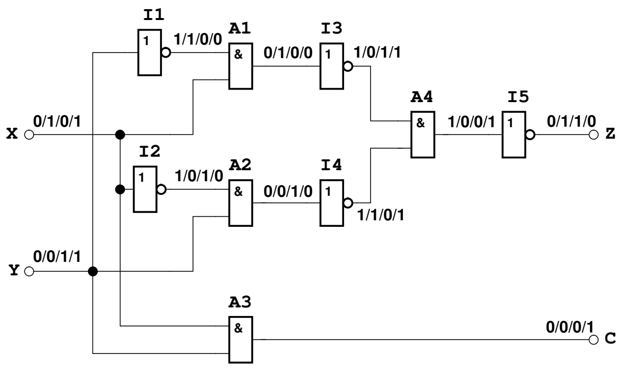

Figure 3: A Ripple-Carry Adder

```
equal(A,B)
  (A >= B) and (B > A)
end equal
```

Assume further the existence of the following test cases:

A	B	Output	Expected Output
1	2	false	false
2	1	false	false
1	1	false	true

Only the third input vector (A = 1,B = 1) leads to a wrong output value (false instead of true) and the following single diagnoses:

1. The `and` function is faulty.
2. The condition `B > A` contains the fault.

By using a similar argument as for the adder example, we can show that the function `and` cannot be the source of the bug.

First, assume that the statement explains the fault, e.g., $ab(\texttt{and})$. Then there must exist a replacement, e.g., another function, repairing the program. Since all other expressions are assumed to behave correctly, we can calculate the inputs and output of the `and` function.

A	B	I1	I2	Expected Output
1	2	false	true	false
2	1	true	false	false
1	1	true	false	true

Such a replacement function must deliver true and false using the same input vector! Therefore no replacement can be found. Contradicting our initial assumption. As final result we get that $ab(\texttt{and})$ cannot be a (single) diagnosis. Only the diagnosis $ab(\texttt{B > A})$ remains valid.

A collection of test cases can help to focus on the right fault in a more traditional way as well. Consider for example the recursive function `myMult` implementing the multiplication on natural numbers (including zero).

```
myMult(A,B)
  if (A <=1) then 0
  else B+myMult(A-1,B) end if
end myMult
```

This implementation contains a (single) bug, namely `(A <= 1)`, which should be `(A <= 0)`. Applying five test cases to the implementation results in the following:

Test case	A	B	Expected Output	myMult(A,B)
tc_1	0	2	0	0
tc_2	1	2	2	**0**
tc_3	2	2	4	**2**
tc_4	2	0	0	0
tc_5	2	1	2	**1**

Using program debugging based on program traces (one of the approaches described in (SW96)) leads only to the unsatisfactory result that all subexpressions can explain the faulty behavior. This holds especially for the third test case (tc_3). For tc_2, only the subexpressions leading to the expression `0` are diagnoses. By computing the intersection of all minimal (non-empty) diagnoses, we obtain the diagnoses explaining all test cases. The prerequisite for using the intersection is of course the assumption that all test cases only exhibit one fault in the program. In the general case, the following definition can be used instead together with a general diagnosis algorithm, e.g., Reiter's Hitting Set algorithm.

Formally, a diagnosis Δ for a diagnosis system $(SD, COMP)$ using multiple test cases OS is a subset of $COMP$ such that

$$\forall_{OBS \in OS} \left(\begin{array}{l} SD \cup OBS \cup \{ab(C) | C \in \Delta\} \cup \\ \cup \{\neg ab(C) | C \in COMP \setminus \Delta\} \end{array} \not\models \bot \right)$$

Similarly, a conflict CO is a subset of $COMP$ such that

$$\exists_{OBS \in OS} \left(SD \cup OBS \cup \{\neg ab(C) | C \in CO\} \models \bot\right)$$

A conflict contradicts at least one observation. The above definitions can now be used for computing diagnoses as described in (Rei87; GSW89). The filter criterion previously described in this section, which avoids replacements that are impossible due to contradictory input-output vectors, must then be incorporated into the system description SD, e.g., by introducing meta rules of the form

```
contradiction :- ¬repairable(Diagnosis).
```

In (SW96), we have presented as one of the main differences between software and hardware diagnosis the property that in the software domain the system description is of necessity derived from the program code itself, and therefore describes the *incorrect* behavior of the system. This is the reason why fault modes are represented by replacements, i.e., actually removing components instead of altering their behavior.

The same duality can be found here, with the non-replacability described above being the dual concept to the notion of physical impossibility in hardware diagnosis. The major difference is the infinite nature of the replacement search space as opposed to the fact that physical impossibility is a fixed restriction that always holds. The reduction in the search space does not result from properties of the original expression (diagnosis component), but of the specific replacement expression chosen.

Conclusion

In this paper we have introduced a logic model for the debugging of the functional language EXP. The model includes the handling of recursive functions. Infinite loops are simply avoided by specifying a maximum recursion level (which is just what programmers do to stop computation of a program that exceeds is expected runtime). We use this representation for the diagnosing EXP programs via replacements for expressions (SW96), which yields the following principal advantages: (1) the number of diagnosis candidates can be reduced, and (2) the program can be automatically repaired. Since diagnosis time increases with the number of component modes, replacements should be only used when necessary, i.e., if too many diagnoses are derived and measurement selection is difficult or not possible. Therefore, we have introduced an algorithm that prefers simpler explanations when conducting replacements and incorporates multiple test cases. We have given examples that show how different tests can help to improve diagnosis results by eliminating diagnosis candidates.

In this context the questions becomes relevant how test-cases are actually generated. The test generation is still an open problem in software engineering research. One approach is currently being examined uses path analyses techniques combined with constraint satisfaction techniques to reduce the search space (GBR98). A different technique is the so-called *mutation testing* which attempts to produce testcases for correct behavior of a certain program part by making local alterations, or mutations to the program. Note that there is a certain similarity to the replacement approach. However, mutation testing works by applying alterations to a correct program in the hope of creating a representative set of incorrect ones (OL94). Those testcase generation techniques are intended to find a faulty behavior rather than locating a specific bug. In our case an incremental test generation algorithm using previous results, i.e., testcases and diagnoses, to find a testcase helping to distinguish between diagnoses should have an improved performance. A classification of testcases from a model-based diagnosis point of view with a rough outline of testcase generation has been given in (McI93).

Future research along the track described in this paper includes an empirical evaluation of the proposed methods together with the development of a heuristics for controlling the debugging process. Overall control would consist of selecting an appropriate sequence of actions leading to a bug free program within a minimum amount of time. Actions are *observe*, *use-replacement*, and *use-other-testcase*. It is interesting to note that the relative costs for these actions can be (depending on program size and complexity) significantly different from the usual relationships in the hardware domain. (FN92; WN97) provide a basic framework for integrating planning in model-based diagnosis and can be adapted to be used in the software domain.

References

Gregory W. Bond. *Logic Programs for Consistency-Based Diagnosis*. PhD thesis, Carleton University, Faculty of Engineering, Ottawa, Canada, 1994.

Gregory W. Bond. Top-down consistency based diagnosis. In *Proceedings on the Seventh International Workshop on Principles of Diagnosis*, Val Morin, Canada, 1996.

G. W. Bond and B. Pagurek. A Critical Analysis of "Model-Based Diagnosis Meets Error Diagnosis in Logic Programs". Technical Report SCE-94-15, Carleton University, Dept. of Systems and Computer Engineering, Ottawa, Canada, 1994.

Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proc. IJCAI*, pages 1494–1499, Chambery, August 1993. Morgan Kaufmann Publishers, Inc.

Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

Gerhard Friedrich and Wolfgang Nejdl. Choosing observations and actions in model-based diagnosis-repair systems. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 489–498, Cambridge, MA, October 1992. Morgan Kaufmann Publishers, Inc.

Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. In *Proc. ECAI*, Budapest, August 1996.

Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 53–62, March 1998.

Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

Sheila McIlraith. Generating tests using abduction. In *Proceedings of the Fourth International Workshop on Principles of Diagnosis*, September 1993.

Jefferson A. Offutt and Stephen D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, 1994.

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

Peter Struss and Oskar Dressler. Physical negation — Integrating fault models into the general diagnostic engine. In *Proceedings* 11^{th} *International Joint Conf. on Artificial Intelligence*, pages 1318–1323, Detroit, August 1989.

Ehud Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts, 1983.

Markus Stumptner and Franz Wotawa. A model-based approach to software debugging. In *Proceedings on the Seventh International Workshop on Principles of Diagnosis*, Val Morin, Canada, 1996.

Markus Stumptner and Franz Wotawa. Diagnosing tree-structured systems. In *Proceedings* 15^{th} *International Joint Conf. on Artificial Intelligence*, Nagoya, Japan, 1997.

Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

Brian C. Williams and P. Pandurang Nayak. A reactive planner for a model-based executive. In *Proc.* 15^{th} *IJCAI*, pages 1178–1185, 1997.

Franz Wotawa. *Applying Model-Based Diagnosis to Software Debugging of Concurrent and Sequential Imperative Programming Languages*. PhD thesis, Technische Universität Wien, 1996.