On the Relationship between Model-based Debugging and Program Mutation*

Franz Wotawa
Technische Universität Wien
Institut für Informationssysteme
Database and Artificial Intelligence Group
Favoritenstraße 9–11, A-1040 Wien, Austria
Email: wotawa@dbai.tuwien.ac.at

Abstract

In this paper we discuss the relationship between program mutation and model-based diagnosis applied to locate and fix bugs in programs. We show that the time required to search for single faults is smaller in model-based diagnosis than in general program mutation. We use this result to speed up program mutation for debugging and develop a procedure that combines both approaches. Moreover, we show that a suitable model can deliver results that are as expressive as the results obtained by program mutation. This is done by introducing a fault mode $REPL$ for components associated with statements and expressions.

1 Introduction

In the last decades several techniques have been proposed for supporting or even automating the software debugging process including [Sha83; Wei82; Wei84; Kor88; Kup89; Mur88] or more recent [Jac95; FN94; Nil98; OL94; BH93; BH95]. Some of the techniques use syntactical characteristics, some make use of the program's semantics, and some consider probability measures. Beside the differences in using available knowledge about a program, the techniques vary in being applicable in a given situation, e.g., debugging of small versus large programs, or helping novices versus programming experts in finding bugs. See [Duc93] for a comparison of different approaches to automated software debugging. Although, the proposed techniques have their merits, no general theory comprising all techniques is available, and there is space for optimizations, including the reduction of required knowledge in order to distinguish different bug candidates. To avoid this problems [CFD93] adapts model-based diagnosis (MBD) [Rei87; dKW87] for debugging and compares it with [Sha83]. The authors claim that using MBD results in a reduction of the number of required interactions with an oracle to identify a bug. The oracle, e.g., the user of a debugging tool, or a (formal) specification, is assumed to know the correct behavior of the program and is assumed to correctly answer all questions which are provided by the debugger in order to distinguish bug candidates. [BP94; Bon94] improves and clarifies some notations of [CFD93]. Based on [CFD93] the application of MBD for debugging large programs written in a real-world programming language was introduced by [FSW99]. [SW99] described the model space for debugging together with assumptions about models that has to be taken into account.

In this paper we introduce a model for a small programming language J with a Java-like syntax (see figure 1) and semantics but ignoring object-oriented features. We compare the outcome of the model-based approach with results obtained by using general program mutation for debugging. The program mutation technique presented in this paper based on the ideas of mutation testing [OL94]. We show that MBD requires less time for computing fault locations in the average, and that MBD comprises program mutation if we restrict the size of the considered mutants to a predefined bound. Taking this results together with observations from [FSW99], showing that program slicing [Wei82; Wei84] is related to model-based debugging using a dependency model, we get more evidence that MBD can serve as a general theory for debugging.

In this paper we understand debugging as the correction of a detected misbehavior. This is done by locating and correcting the parts of the program causing a behavior that contradicts at least one test case. Because of the requirement that a program must be executable, syntactical errors usually caught during compiling the program, are out of the scope of this paper. In addition we reduce our view to programs that halt (at least) for the predefined test cases. Although, the described techniques might be capable of providing a solution once an infinite loop has been detected. We call the remaining class of bugs behavioral faults or bugs. This class can be further subdivided into structural and functional bugs. Structural bugs are due to missing variables or wrongly used variables. Functional bugs are faults related to a wrongly used constant, function, or predicate within an expression. Other classes of bugs include missing statements, statements that should be eliminated, or a wrong sequence of statements. The latter classes of bugs require a larger corrections of the original program than replacing one function by another. In this paper we ignore those kinds of bugs and consider only behavioral bugs related to wrong variables, constants, functions, and predicates. Note, that the same restriction of the con-

*This work was partially supported by Austrian Science Fund projects P12344-INF and N Z29-INF.

program ::= { *statement* }*

statement ::= *assignment* | *conditional* | *while*

assignment ::= *Id* = *expr* ;

conditional ::=
if (*expr*) { { *statement* }* }
[{ { *statement* } * }]

while ::= **while**(*expr*){ { *statement* }* }

expr ::= *Id* | *Const* | (*expr op expr*) | (*mop expr*)

op ::= + | - | * | / | < | > | <= | >= | == | **and** | **or** | ...

mop ::= + | - | **not** | ...

Figure 1: Syntax of the language J **Remark:** Variables are emphasized. Keywords of the language are bold typed. All terminal symbols are underlined. The meta-characters have the following meaning: A '*' stands for zero or more iterations, a '+' means more than zero iterations, the braces '{','}' indicate a block, and brackets '[',']' an optional block.

sidered kinds of bugs are also used in other more traditional debugging techniques.

2 The debugging problem

Formally, the debugging problem is characterized by a given program Π and a set of test cases T. The program Π must be written according to the syntax which is depicted in Figure 1. We define the syntax of J as simple as possible to avoid overhead in the definitions. Therefore, without restricting generality user-defined functions, and general functions are not considered. The semantics of Π (as for other sequential languages) can be formally described using a function $eval$, that maps programs (statements) and variable environments to other variable environments. A variable environment env is itself a function that maps variables to their values. The evaluation function $eval$ for J is defined as follows:

Statements

$$eval(\{st_1,\ldots,st_n\},env) = eval(\{st_2,\ldots,st_n\},eval(st_1,env))$$

$$eval(\{\},env) = env$$

Assignments

$eval(\texttt{X = E},env) = env'$ with

$$env'(v) = \begin{cases} env(v) & \text{if } v \neq \texttt{X} \\ eval(\texttt{E},env) & \text{otherwise} \end{cases}$$

Conditionals

- with *else* branch:
$$eval(\texttt{if E TSt ESt},env) = \begin{cases} eval(TSt,env) & \text{if } eval(E,env) = true \\ eval(ESt,env) & \text{if } eval(E,env) = false \end{cases}$$
- without *else* branch:
$$eval(\texttt{if E TSt},env) = \begin{cases} eval(TSt,env) & \text{if } eval(E,env) = true \\ env & \text{otherwise} \end{cases}$$

While statements

$$eval(\texttt{while E St},env) = \begin{cases} eval(\texttt{while E St},eval(St,env)) & \text{if } eval(E,env) = true \\ env & \text{if } eval(E,env) = false \end{cases}$$

Expressions

- **Variables** $eval(\mathrm{V},env) = env(\mathrm{V})$
- **Constants** $eval(\mathrm{C},env) = C$
- **Other expressions**
$eval((\mathrm{E1\ op\ E2}),env) = op(eval(\mathrm{E1},env),eval(\mathrm{E2},env))$
$eval((\mathrm{op\ E}),env) = op(eval(\mathrm{E},env))$

After having defined the syntax and semantics of J we formally define test cases. A test case for a program must specify the correct behavior. In this sense a test case is a small part of the specification that maps an input state to the expected output state. In our case states correspond to variable environments. So, a test case is a tuple (I,O) where I is the environment before executing a program Π and O is the expected environment to be delivered when executing the program. We say that a program pass a test iff $eval(\Pi,I) = O$. Otherwise, we say that a program is buggy. In a real-world setting not only one test case is used to check a program. Therefore, we assume a function $tests$ mapping programs to a set of test cases.

We illustrate the definitions using a small example program *demo* comprising three assignments:

```
1.   x = ( a * b ) ;
2.   min = ( x * lb ) ;
3.   max = ( x * ub ) ;
```

We use this program as running example of this paper. Program *demo* takes 4 variables *a*, *b*, *lb*, and *ub* as input and computes values for the variables *min* and *max*. The variable *x* is used as intermediate variable to avoid recomputation of `a*b`. The program *demo* is for computing a minimum and maximum value for `a*b` using a lower bound *lb* and an upper bound *ub* value. In order to test the program we have to specify test cases, e.g.,

$$tests(demo) = \{(\{env(a)=1,env(b)=2,env(lb)=1,env(ub)=2\}, \{env(min)=2,env(max=4)\})\}$$

Using this test case and executing the program according to its semantics leads to the conclusion that *demo* is correct with respect to the given test cases. Now consider the case where line 3 of *demo* is modified to:

```
3.   max = ( x / ub ) ;
```

We refer the wrong version of *demo* as *demo2*. Using the same test cases as before, i.e., $tests(demo2) = tests(demo)$ leads to the conclusion that the program *demo2* must contain a bug. Because the execution of *demo2* leads to $env(max) = 1$ which obviously contradicts the expected value $env(max) = 4$. In this case the programmer (user) would be interested in locating the source of this misbehavior. Using a traditional debugger, that guides the user in a sequential manner through the source code, finding the bug in line 3 requires 3 steps. Each of which consists of asking the programmer implicitly or explicitly about variable values. Implicitly means that the variables are presented to the user but where user interaction, e.g., answering a question about correctness or not, is not required. An explicit question must be answered in order to go to the next line in the code. Traditional debuggers are quite effective if the bug is located at the beginning of a program fragment to be debugged. However, in cases where the bug is located at the end, debuggers are ineffective. In practice this effect is tried to be reduced by using break-points in programs. The debugger evaluates the program up to a break-point and starts interacting with the user afterwards. This technique is good for minimizing user interaction. But in order to place the break-point at the right position a good understanding of the structure and behavior of at least parts the program is required.

Another way of avoiding too many steps using a debugger is program slicing [Wei82; Wei84] or similar techniques using dependencies between variables, e.g., [FSW99]. Such dependency oriented methods collect those statements that may have an influence on a given variable. In the example *demo2* only the statements in line 1 and 3 have an influence on variable *max* whereas this is not the case for the statement in line 2. An intelligent debugger can make use of this knowledge and focus the user's attention to the two relevant statements. As a result the number of required steps is reduced. However, such a reduction cannot always be provided by program slicing. In order to further reduce the number of bug candidates, i.e., statements or expressions, the full programming language semantics must be used. For example consider that line 1 of program *demo2* is faulty and the other statements are not. From $env(min) = 2$ and $env(lb) = 1$ we can derive that $env(x)$ must be 2. Taking this result together with $env(ub) = 2$ we derive $env(max) = 1$ that again contradict our expected value. Hence, statement 1 cannot be the single source of the misbehavior. Using similar arguments, we get line 3 as the only single bug location. If we prefer minimal solutions, only line 3 of *demo* would be presented to the user.

3 Program mutation and debugging

Mutation testing, e.g., [BDLS80; OL94], was originally developed for constructing a set of tests which distinguish between a given program and any nonequivalent program and for measuring the quality of test cases. The nonequivalent program is generated using the given one and applying the mutation transformation. The ideas of mutation testing can be used for debugging. In [Duc93] the author classifies program mutation as sort of filtering technique for debugging[1]. The changed parts of a program are bug candidates, if the modified version of the program, i.e., the mutant, passes all test cases. This reduces the search for bug locations to the computation of mutants passing the test cases. Moreover, the mutants not only provide bug locations but also repair suggestions.

In the following we define program mutation formally, give an algorithm for debugging, and show some properties of the algorithm. For presentation reasons we restrict mutation transformations to expressions. Statements cannot be mutated directly. They only can be modified by changing the used expressions. A valid mutation transformation (E, N) maps an expression E to a different expression $N \neq E$. We distinguish 3 types of mutations: (1) replacing, (2) eliminating, or (3) adding a constant, variable, or function. Here is an example for a mutation transformation for the different types:

$$(\mathtt{x\ +\ y}, \mathtt{x\ -\ y}) \quad (1)$$
$$(\mathtt{x\ +\ y}, \mathtt{x}) \quad (2)$$
$$(\mathtt{x}, \mathtt{x\ +\ y}) \quad (3)$$

Given a program Π and a transformation (E, N) we can define a mutant Π' as a program obtained by replacing the expression $E \in \Pi$ with expression N. Note that every expression used in a program can be uniquely identified. From here on we make use of the notation E^i to distinguish expressions whenever necessary, e.g., two expressions $\mathtt{x}^1$ and $\mathtt{x}^2$ that are used by another expression $\mathtt{x}^1 + \mathtt{x}^2$. We write $(\Pi, < (E, N) >)$ to represent the mutant Π'. If (E, N) is a valid transformation Π' must be syntactically different to Π. The new mutant is a program that can be mutated again. $(\Pi, < (E_1, N_1), \ldots, (E_n, N_n) >)$ denotes a sequence of n mutations of a program Π with the following meaning: First the transformation (E_1, N_1) is applied, then (E_2, N_2) on the resulting mutant, and so on. During the rest of this paper we do not distinguish between a single mutant and a sequence of mutations. Both are called a mutant. We define a mutation diagnosis, i.e., a bug candidate, as follows:

Definition 3.1 (Mutation Diagnosis) *Let Π be a program and $tests(\Pi)$ a set of test cases for Π. A mutant $\Pi' = (\Pi, < (E_1, N_1), \ldots, (E_n, N_n) >)$ is a mutation diagnosis (or mutation bug) iff Π' passes all test cases $tests(\Pi)$.*

Mutation diagnoses can be ordered wrt. the differences between the mutant and the original program. This difference is given by the sum of the size differences of the applied expression modifications. We define the difference δ between two expression as follows:

- For all expressions E: $\delta(E, E) = 0$
- For variables and constants $X \neq Y$: $\delta(X, Y) = 1$
- For an expression E with $size(E) > 1$, a variable or constant X, and an operator $\oplus$:
  $$\delta(E \oplus X, E) = 1$$
  $$\delta(X \oplus E, E) = 1$$
  $$\delta(E, E \oplus X) = 1$$
  $$\delta(E, X \oplus E) = 1$$
  $$\delta(E, X) = size(E) + 1$$
  $$\delta(X, E) = size(E) + 1$$
- For expressions E_1, E_2, E_3, E_4 and operators $\oplus_1 \neq \oplus_2$: $\delta(E_1 \oplus_1 E_2, E_3 \oplus_1 E_4) = \delta(E_1, E_3) + \delta(E_2, E_4)$ and $\delta(E_1 \oplus_1 E_2, E_3 \oplus_2 E_4) = 1 + \delta(E_1, E_3) + \delta(E_2, E_4)$

[1] Although, [Duc93] claims that model-based diagnosis is element of the program mutation category, we think that this is not the case. For example [CFD93] compares model-based diagnosis with algorithmic debugging [Sha83] which is accordingly to [Duc93] part of the algorithmic behavior verification category.

We define the size of an expression as follows:

$$size(E) = \begin{cases} 1 & \text{if } E \text{ is a variable or constant} \\ size(E_1) + size(E_2) + 1 & \text{if } E = E_1 \oplus E_2 \end{cases}$$

An orthogonal concept of ordering diagnoses wrt. the differences between the mutant and the original program is to order (and limit) diagnosis candidates wrt. minimality. A mutation diagnosis Δ is a minimal diagnosis iff there exists no other mutation diagnosis Δ' that is a subset of Δ. Similar to the hardware domain we are interested in finding minimal bugs. For example, the mutation diagnosis $< (\texttt{x/ub}^3, \texttt{x*ub}^3) >$ is a minimal diagnosis for program *demo2*, whereas $< (\texttt{x/ub}^3, \texttt{x*ub}^3), (\texttt{x*lb}^2, \texttt{x*ub}^2), (\texttt{x*ub}^2, \texttt{x*lb}^2) >$ is not. E^i denotes the expression E used in statement i.

In the following section we describe the algorithm MUTATION for computing all possible mutations of a program Π up to a given maximum difference MAX.

3.1 Computing mutants

The MUTATION algorithm computes all possible mutations for a given program Π in an iterative way. The algorithm uses a set of mutations generated one iteration cycle before. In the first cycle only the original program is considered. For each of the previously generated programs (or the original program) MUTATION selects a statement and applies the basic mutation functions resulting in a new mutant considered in the next iteration. The selection process is finished when all statements have been taken into account. If we use the algorithm MUTATION we obtain a sequence of all possible mutants of Π up to the size MAX. The sequence is ordered by the number of changes, i.e., the size of the mutants.

Algorithm MUTATION(Π, MAX)

1. Let i be 0, and S_i be a set containing the unmutated original program $(\Pi, <>)$.
2. Let S_{i+1} be the empty set.
3. For all elements $m = (\Pi, N)$ in S_i do the following:
   (a) For all statements c in m call the algorithm BASIC_MUTATIONS with Π, c, and N as arguments. BASIC_MUTATIONS returns a set of mutants M for c. Add all elements of M to S_{i+1}.
4. If i is not greater than MAX, increase i by 1 and goto 2. Otherwise, return the sequence $S_0, \ldots, S_{MAX}$ as result.

MUTATION calls the algorithm BASIC_MUTATIONS for computing mutants of single statements. The algorithm distinguishes the three kinds of statements, i.e., assignments, conditionals, and loops, and takes the appropriate actions. Mutations for assignments include changing the target variable, and mutating the expression. Mutants for conditionals may vary in the expression or in a mutated statement in one of the two statement blocks. Loops have mutants with changed conditional expressions or differences in exactly one statement. BASIC_MUTATIONS requires 3 arguments: the program Π, a statement c, and a mutant N that was generated in a previous step.

Algorithm BASIC_MUTATIONS(Π, c, N)

1. Let M be the empty set.
2. If c is an assignment statement of the form `X = E` then do the following:
   (a) For all variables $Y \neq X$ add the mutant $(\Pi, N \cup < (X, Y) >)$ to M.
   (b) To generate mutations for E call EXPR_MUT with Π, E, and N as arguments and add the resulting mutants to M.
3. If c is a conditional statement then compute the mutations for the conditional expression using EXPR_MUT and add the result to M. Afterwards, compute all mutants for the statements in both branches of the conditional using BASIC_MUTATIONS and add all of them to M.
4. If c is a loop statement, compute the mutations for the conditional expression using EXPR_MUT and add the result to M. In the next step compute all mutants for the statements in the block using BASIC_MUTATIONS. Add the resulting mutants to M.
5. Return the set M of mutants.

The EXPR_MUT algorithm compute mutants with exactly one difference between them and the given expression according to the function δ. Variables and constants are replaced by other variables or other constants. Operators are to be changed by other operators, and expressions with an operator are reduced to one argument. In addition EXPR_MUT allows for adding an additional operator to an existing one. EXPR_MUT requires 3 arguments: the program Π, an expression E to be mutated by EXPR_MUT, and a mutant N.

Algorithm EXPR_MUT(Π, E, N)

1. Let M be an empty set.
2. If E is a variable or constant then for all variables or constants $Y \neq E$ add the mutant $(\Pi, N \cup < (E, Y) >)$ to the set M. Add mutants of the form $(\Pi, N \cup < (E, \texttt{E} \oplus \texttt{X}) >)$ to M where $\oplus$ is an operator, and X is a variable or constant.
3. If E is of the form $\texttt{X} \oplus \texttt{Y}$ then for all operators $\oplus' \neq \oplus$ add $(\Pi, N \cup < E, N >)$ to M where N is one of the following elements: $\texttt{X} \oplus' \texttt{Y}$, `Y`, `X`.
4. Return M.

The MUTATION algorithm computes all possible mutants for a program up to a given size. However, it is possible that equal mutants are computed with different size. For example consider a program Π with original expression `X - Y` and its mutant `X + Y`. Two possible mutations can lead to the same mutant. The first is $(\Pi, < (\texttt{X-Y}, \texttt{X+Y}) >)$, and the second is $(\Pi, < (\texttt{X-Y}, \texttt{X}), (\texttt{X+Y}, \texttt{X}) >)$. Hence, MUTATION does not necessarily provide a minimal sequence of possible mutants.

3.2 Computing mutation diagnoses

Mutation diagnoses (or bugs) are mutants passing all given test cases. Therefore, the algorithm MUTATION can be used for computing all candidates that are checked against the test cases. The remaining mutants are bug candidates. The following algorithm MUTATION_DIAG computes all minimal cardinality diagnoses for a given program Π and set of test cases $T = tests(\Pi)$.

Algorithm MUTATION_DIAG(Π, T)

1. Let D be an empty set and let i be 1.
2. *Compute candidates* Compute all mutants of size i by calling MUTATION(Π, i). Store the mutants in D.
3. *Check candidates* For all $\Delta \in D$ check whether Δ passes the test cases T or not. If Δ does not pass, then remove Δ from D.
4. If D is empty, increase i by one and goto 2. Otherwise return D as result.

It is obvious that in general MUTATION_DIAG is sound and complete wrt. the given definitions and assumptions. The more interesting question is the time and space complexity of MUTATION_DIAG. The time complexity T_M depends on the number of test cases t, the maximum runtime r for checking one mutant against one test case, the number of mutants m and the time g required for creating one mutant, i.e., $T_M \sim O(t \cdot r \cdot m + m \cdot g)$. The time required for creating one mutant must linearly depend on the size of the program Π, i.e., $g \sim O(e + s)$, where s denotes the number of statements and e the number of expressions of Π. The number of created mutants depends also on the size of the program and the maximum difference MAX. In each iteration the number of mutants is equal to the number of mutants in the previous iteration multiplied by a constant factor. In the first iteration the number of mutants is equal to the program's size multiplied by the factor. Hence, we have $m \sim O((e+s)^{MAX})$. In summary we get the following result for the time complexity.

Theorem 3.1 (Time complexity of mutation diagnosis) *The time complexity T_M for computing all mutation diagnoses up to a given size MAX is* $T_M \sim O(t \cdot r \cdot (e+s)^{MAX} + (e+s)^{MAX+1})$.

The space complexity of mutation diagnosis can be computed in a similar way. For each mutant there must be a reserved space of size $O(MAX)$. In addition we require space s_E for executing the mutant. In summary the space requirements depend on the required space s_E and a space for storing all mutants.

Theorem 3.2 (Space requirements of mutation diagnosis) *The space complexity S_M for computing all mutation diagnoses up to a given size MAX is* $S_M \sim O(s_E + MAX \cdot (e+s)^{MAX})$.

The time and space complexity of mutation diagnosis is not a problem in practice if we restrict the size to be small. In most cases the search for mutation diagnoses is restricted to $MAX = 1$ and at most to 2 or 3. In the following sections we show how mutation diagnosis can be enhanced by using model-based reasoning.

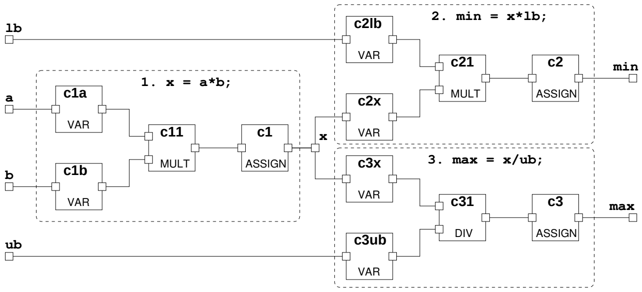

Figure 2: Graphical representation of the *demo2* model

4 Model-based debugging

In various papers the authors' have shown that model-based diagnosis can be used for debugging, e.g., [CFD93; FSW99; SW99; MSW00; Wot00]. Since this paper aims not in presenting a new model for a sequential programming language, we use the value-based model of [MSW00] which was developed for a subset of Java. To be self contained we briefly summarize the model. For more details we refer to [MSW00]. The logical model represents the semantics of the statements of our language: assignments, conditionals, and loops. Each statement and expression is associated with an diagnosis component. The behavior of the components captures the semantics of the underlying statement or expression. The variables are represented by connections. In addition we have connections between expression and its subexpressions and whenever required, e.g., if a statement has an expression. For assignments, expressions, and conditionals we have 2 modes: AB and $\neg AB$ with the usual meaning. Loops have the same modes plus 1 additional mode, the *Loop* mode restricting the execution of the block statements to a fixed number. This number is assumed to depend on the test case.

For example the logical model SD_{demo2} of our *demo2* program is given by the following sentences:

- Component behavior:

$$\begin{gathered} ASSIGN(C) \Rightarrow (\neg AB(C) \rightarrow in(C) = out(C)) \\ MULT(C) \Rightarrow (\neg AB(C) \rightarrow in_1(C) \cdot in_2(C) = out(C)) \\ DIV(C) \Rightarrow (\neg AB(C) \rightarrow in_1(C)/in_2(C) = out(C)) \\ VAR(C) \Rightarrow (\neg AB(C) \rightarrow in(C) = out(C)) \end{gathered}$$

- Structure: Declaring the component's type

$$\begin{gathered} ASSIGN(c1) \wedge ASSIGN(c2) \wedge ASSIGN(c3) \\ MULT(c11) \wedge MULT(c21) \wedge DIV(c31) \\ VAR(c1a) \wedge VAR(c1b) \wedge VAR(c2x) \\ VAR(c2lb) \wedge VAR(c3x) \wedge VAR(c3ub) \end{gathered}$$

and the connections

$$\begin{gathered} out(c1a) = in_1(c11) \wedge out(c1b) = in_2(c11) \\ out(c11) = in(c1) \wedge out(c1) = in(c2x) \\ out(c2x) = in_1(c21) \wedge out(c2lb) = in_2(c21) \\ out(c21) = in(c2) \wedge out(c1) = in(c3x) \\ out(c3x) = in_1(c31) \wedge out(c3ub) = in_2(c31) \\ out(c31) = in(c3) \end{gathered}$$

The graphical representation of the *demo2* model is depicted in Figure 2. We see for example that the assignment statement of line 1 is mapped to component $c1$, the multiplication function is mapped to $c11$, the variable accesses `a`, `b` are mapped to $c1a$ and $c1b$ respectively. In the next step we use SD_{demo2} together with the test case for computing bug candidates. We convert the test case to a set of observations

$$OBS = \left\{ \begin{array}{l} in(c1a) = 1, in(c1b) = 2, in(c2lb) = 1, \\ in(c3ub) = 2, out(c2) = 2, out(c3) = 4 \end{array} \right\}$$

and compute 4 single diagnoses

$$\{AB(c3)\}, \{AB(c31)\}, \{AB(c3x)\}, \{AB(c3ub)\}$$

which can be mapped back to the statements and expressions in the source code of *demo2*. All single diagnoses indicate that the bug can only be in line 3.

4.1 Enhancing mutation diagnosis

In this section we show how model-based diagnosis can be used to speed up mutation diagnosis. The idea behind is to use the results obtained from model-based diagnosis to avoid the computation of unnecessary mutants. Only those expressions lying in at least one diagnosis are subject to be mutated. Moreover, taking a diagnosis and the test cases we can (at least partially) compute the expected behavior of components, i.e., statements and expressions. Table 1 shows the expected values for the single diagnoses computed for *demo2*. Using this information we further can reduce the number of mutations to be considered in order to find a solution. For example, replacing the mutant (`x/ub`, `x-ub`) cannot explain the expected values $(in_1(c31), in_2(c31), out(c31)) = (2, 2, 4)$. Since all of the required tests for eliminating mutations from the list of possible mutants use local information only, they are not so time consuming as executing the whole program after mutating it.

In order to show that model-based diagnosis can really be used to speed up mutation diagnosis, we have to show that computing diagnoses is less time consuming than computing all mutants. We first discuss the time complexity of model-based diagnosis for computing single diagnosis. In the worst case all $e + s$ diagnosis components must be checked for being faulty. Every check lasts $O(r)$, where r is the maximum runtime of the program for the test cases. Since we have t different test cases the time complexity is given as follows:

Theorem 4.1 (Time complexity - single diagnoses) *The time complexity for locating single bugs in a program* Π *using model-based diagnosis is* $\sim O(t \cdot r \cdot (e + s))$.

In general the time complexity depends on the number of subsets to check. Hence, we have $O(t \cdot r \cdot 2^{(e+s)})$ when assuming that every component has about 2 modes, e.g., AB or $\neg AB$. Although this is not the case for our model in general, it is a good approximation. Because only loop statements have a (very) limited number of additional modes, and usually there is only a vanishing number of loops in a program (if compared to the number of expressions). For mutation debugging providing a mutant for all statements and expressions requires a MAX to be $e + s$. Hence, the time complexity in this case is $O(t \cdot r \cdot (e + s)^{(e+s)} + (e + s)^{(e+s+1)})$ which is obviously larger than $O(t \cdot r \cdot 2^{(e+s)})$. Space complexity for computing all single diagnoses is of order $O(e + s)$, if we assume that every component has only 2 modes. The space complexity is also smaller than the space complexity of mutation debugging.

In summary the enhancement of mutation debugging include 3 steps: First, compute diagnoses using model-based diagnosis and a value-based model. Second, compute mutants only for components that are element of at least one diagnosis. Third, use the locally available expected values for eliminating mutants instead of executing the whole program.

4.2 Expressing mutation diagnosis

In this section we discuss how mutation diagnosis can be expressed in terms of model-based diagnosis. The straight forward way is do use a predicate $REPL$ having 2 arguments (instead of 1) for each diagnosis component. The first argument is for the component identifier and the second argument represents the desired behavior of the component. For example, we can declare the replacement fault mode $REPL(or)$ to an *and* function:

$$AND(C) \Rightarrow (REPL(C, or) \rightarrow out(C) = in_1(C) \vee in_2(C))$$

Other replacement modes can be easily declared in the same manner. For variable access or assignment components such a replacement mode is not so easy to define. This is due to the fact that replacing a variable by another variable has an impact on the structure of the associated diagnosis system. We can define a replacement mode for a variable access components. The respective diagnosis component must have an input for each variable used in the program. Depending on the replacement mode the corresponding input value is propagated to the output and the output value is propagated to the corresponding input. Such a diagnosis component can be seen as a select function. A similar technique can be used for assignment statements, where the output connection must vary according to its $REPL$ mode that corresponds to different target variables.

The advantage of the $REPL$ mode is that it provides a possible repair action, i.e., replacing the corresponding expression with the specified expression. Although, we can introduce arbitrary replacement modes, they require additional connections between diagnosis components and cause the overall diagnosis time to increase significantly. Moreover, a more sophisticated change of the program, e.g., replacing `x + y` by `x + y - (z * w)`, can be hardly expressed by our model. In this case the diagnosis component corresponding to the *plus* operator must have knowledge about the other variables `z` and `w`. Hence, we have to introduce new inputs not only for the variable access components but also for all other components. Because of these problems we suggest to use replacement modes whenever we know a small fraction of the components and expressions to be part of a highly probable diagnosis candidate.

Diagnosis	in(c3x)	out(c3x)	in(c3ub)	out(c3ub)	out(c31)	out(c3)
$\{AB(c3)\}$	2	2	2	2	1	4
$\{AB(c31)\}$	2	2	2	2	4	4
$\{AB(c3x)\}$	2	8	2	2	4	4
$\{AB(c3ub)\}$	2	2	2	0.5	4	4

Table 1: Expected values for *demo2*

5 Conclusion

This paper dealt with establishing the relationships between model-based diagnosis and program mutation to be used in software debugging. In the first part of this paper, we discussed the concept of program mutation for debugging, and we presented definitions and algorithms for computing mutants of a given size. Moreover, we elaborated the time and space complexity of mutation diagnosis. In the second part, we briefly discussed how to use model-based diagnosis for debugging and tried to investigate the relationships between the two approaches. The main results of this paper are:

- Mutation testing can be significantly enhanced by using model-based diagnosis. The resulting diagnoses are seen as focus for mutation diagnosis. Only mutants of expressions and components that are element of at least one diagnosis are created.
- In principle the advantage of mutation diagnosis, i.e., the computation of possible corrections, can be adopted for the model-based framework. A replacement mode $REPL$ maps different behaviors to diagnosis components. This mode expresses changes of expressions directly. However, beside the caused increase of diagnosis time, the complexity of the resulting model increases as well.

In future debugging systems the combination of model-based diagnosis techniques and program mutation seems to be a good choice. Whereas the first supports the discrimination between statements that may cause a misbehavior, the latter provides repair suggestions.

References

[BDLS80] Timothy Budd, R. DeMillo, R. Lipton, and F. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proc. Seventh ACM Symp. on Princ. of Prog. Lang. (POPL)*. ACM, January 1980.

[BH93] Lisa Burnell and Eric Horvitz. A Synthesis of Logical and Probabilistic Reasoning for Program Understanding and Debugging. In *Proceedings of the International Conference on Uncertainty in Artificial Intelligence*, pages 285 –291, 1993.

[BH95] Lisa Burnell and Eric Horvitz. Structure and Chance: Melding Logic and Probability for Software Debugging. *Communications of the ACM*, pages 31 – 41, 1995.

[Bon94] Gregory W. Bond. *Logic Programs for Consistency-Based Diagnosis*. PhD thesis, Carleton University, Faculty of Engineering, Ottawa, Canada, 1994.

[BP94] 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.

[CFD93] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1494–1499, Chambery, August 1993.

[dKW87] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[Duc93] Mireille Ducassé. A pragmatic survey of automatic debugging. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging, AADEBUG '93*, Springer LNCS 749, pages 1–15, May 1993.

[FN94] Peter Fritzson and Henrik Nilsson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3), 1994.

[FSW99] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(2):3–39, July 1999.

[Jac95] Daniel Jackson. Aspect: Detecting Bugs with Abstract Dependences. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.

[Kor88] Bogdan Korel. PELAS–Program Error-Locating Assistant System. *IEEE Transactions on Software Engineering*, 14(9):1253–1260, 1988.

[Kup89] Ron I. Kuper. Dependency-directed localization of software bugs. Technical Report AI-TR 1053, MIT AI Lab, May 1989.

[MSW00] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Modeling Java Programs for Diagnosis. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, Berlin, Germany, August 2000.

[Mur88] William R. Murray. *Automatic Program Debugging for Intelligent Tutoring Systems*. Pitman Publishing, 1988.

[Nil98] Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linköping University, April 1998.

[OL94] Jefferson A. Offutt and Stephen D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, 1994.

[Rei87] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[Sha83] Ehud Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts, 1983.

[SW99] Markus Stumptner and Franz Wotawa. Debugging Functional Programs. In *Proceedings* 16^{th} *International Joint Conf. on Artificial Intelligence*, Stockholm, Sweden, August 1999.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[Wot00] Franz Wotawa. Debugging VHDL Designs using Model-Based Reasoning. *Artificial Intelligence in Engineering*, 14(4):331–351, 2000.