Using distinguishing tests to reduce the number of fault candidates

Mihai Nica [1], **Simona Nica** [1], **and Franz Wotawa** [1]

[1] *Technische Universitä Graz, Institute for Software Technology, Inffeldgasse 16b/II, 8010, Austria.*
{*mnica, snica, wotawa*}*@ist.tugraz.at*

ABSTRACT

Tools for automated fault localization usually generate too many bug candidates depending on the underlying technique. Hence, further information is required in order to further restrict the bug candidates. Approaches that rely on specific knowledge of the program to be debugged like variable values at specific position in the source code, are not easily accessible for users especially in case of software maintenance. In order to avoid this problem we suggest to integrate testing for restricting the number of bug candidates. In particular, we suggest to compute possible corrections of the program and from this, distinguishing test cases. A distinguishing test case is a test that reveals different output values for two given program variants, given the same input values. Besides the formal definitions, and algorithms we present the first empirical results of our approach. The use of mutations and distinguishing test cases substantially reduces the number of bug candidates.

1 INTRODUCTION

Debugging, i.e., detecting, locating, and correcting a bug, in a program is considered a hard and time consuming task. This holds especially in case of software maintenance where the programmer has little knowledge of the program's structure and behavior. Today's research activities mainly focus on the fault detection part of debugging. Automated verification and testing methods based on models of the system and specification knowledge have been proposed. Little effort has been spent in automated fault localization and even less in fault correction. There has been also no research activity bringing together testing and fault localization and correction, except the fact that test cases are used for debugging. However, to the best of our knowledge there is no work that analyzes the impact of test suites on the obtained debugging results.

In this paper, we contribute to the test case generation problem in order to improve the obtained results of automated debugging based on a model of the program. In particular we show how test cases can be generated to distinguish potential diagnosis candidates. A potential diagnosis candidate, or diagnosis candidate for short, is a statement that can explain why the test cases fail. A diagnosis candidate needs not to be the real bug. But the real bug should be included in the list of diagnosis candidates delivered by an automated debugger.

We now consider the following code snippet to illustrate our combined debugging and testing approach. We use this small program to avoid introducing too much technical overhead and to focus on the underlying idea.

```
        ...
1.      i = 2 * x;
2.      j = 2 * y;
3.      o1 = i + j;
4.      o2 = i * i;
        ...
```

We cannot say anything about the correctness of such a code fragment without any additional specification knowledge. Let us assume that we also have the following test case specifying expected outputs for the given inputs: `x = 1, y = 2, o1 = 8, o2 = 4`. Obviously, the program computes the outputs `o1 = 6` and `o2 = 4`, which contradicts the given test case. Therefore, we know that there is a bug in the program and we have to localize and correct it. At this stage we might use different approaches for computing potential fault locations. If using the data and control dependencies of the program, we might traverse the dependencies from the faulty outputs to the inputs backward. In our example, we are able to identify statements 1, 2, and 3 as potential candidates.

A different way to locate bugs is to consider statements as equations and to introduce correctness assumptions. If the test case together with the assumptions and the equations is consistent, the assumptions stating incorrectness of statements can be used as potential diagnosis candidates. Consider for example Statement 1 to be faulty and all other statements to be correct. As a consequence, Statement 1 does not determine a value for variable `i`. However, from Statement 4 and the test case we can conclude that `i` has to be 2 (if assuming only positive integers). Hence, we are

able to compute a value for o1 again, which contradicts the given test case. Therefore, the assumption that Statement 1 is a diagnosis candidate, cannot be correct. Note that even when assuming that i might be -2, we are able to derive a contradiction. It is also worth noting that the described approach for generating diagnosis candidates can be fully automated.

We are able to apply this technique for making and checking correctness assumptions for all statements and finally obtain statements 2 and 3 as diagnosis candidates. For larger programs we might receive a lot of potential diagnosis candidates and the question of how to reduce their number becomes very important. One solution is to ask the user about the expected value of intermediate variables like i or j for specific test cases. Such an approach requires more or less executing the program stepwise. Moreover, especially in the case of software maintenance where a programmer is not very familiar with the program answering questions about values of intermediate variables, this can hardly be done. Therefore, we suggest an approach that computes test cases, which allow to distinguish the behavior of diagnosis candidates. More specifically, we are searching for inputs that reveal a different behavior of diagnoses candidates. Such test cases are called distinguishing test cases (Wotawa *et al.*, 2010). In case no such distinguishing test case can be computed, the diagnosis candidates are, from the perspective of their input output behavior, equally good.

What prevents us from applying the approach of distinguishing test cases to distinguish diagnosis candidates is the fact that the fault localization approaches only give us information about the incorrectness and correctness of some statements but not about the correct behavior of potentially faulty statements. Hence, computing test cases is hardly possible. In order to solve this problem we borrow the idea of mutation or genetic-based debugging (Weimer *et al.*, 2009; Debroy and Wong, 2010). Mutants, i.e., variants of the original program, are computed and tested against a test suite. The mutants that pass all test cases are potential diagnosis candidates. Computing mutants for all statements and testing them against the test suite is very time consuming and some techniques for focusing on relevant parts of the program have been suggested. In our case we are able to use the diagnosis candidates for focusing on relevant parts of the program. Hence, when finding a mutant for a diagnosis candidate that passes all test cases, we do not only localize the bug but also state a potential correction.

For our example program we might obtain two mutations m_1, m_2 for statements 2 and 3, e.g.: m_1 =`2. j=3*y` and m_2 = `3. o1=i+j+2`. Obviously there are more mutations available but for illustrating the distinguishing test cases we only use these two now. A distinguishing test case for these mutants is `x = 1, y = 1`. Mutant m_1 computes the value 5 and mutant m_2 the value 6 for the output variable o1. If we know the correct value of o1, we are able to distinguish the two mutants. From this example we conclude that we are able to distinguish diagnosis candidates using distinguishing test cases. What remains an open research issue is to provide empirical evidence that the approach is feasible and provides a reduction of diagnosis candidates when applied to general programs.

In this paper, we introduce and discuss the approach and tackle the research question regarding the approach's practicability with some exceptions. The programs used for the empirical evaluation are small programs and they mainly implement algebraic computations. Moreover, we do not handle object-oriented constructs. However, we do not claim to answer the research question completely. We claim that the approach can be used for typical programs comprising language constructs like conditionals, assignments, and loops. The structures of the used programs are similar to those of larger programs or at least we do not see why there should be any big differences. Another argument is that the approach is mainly for debugging at the level of methods comprising a smaller amount of statements where our approach is definitely feasible.

```
1.     tmp = (a + 1); // ERROR
2.     if (b == 0) {
3.         result = -1;
        } else {
4.         result = 0;
5.         while (tmp > 0) {
6.             result = result + 1;
7.             tmp = tmp - b;
           }}
```

Figure 1: A program for dividing two natural numbers

The paper is organized as follows. We first introduce the basic definitions. These include the definition of a test case and stating the debugging problem. Since the debugging approach is based on a model of the program, we introduce a constraint representation of programs that serves our purpose in the next section. This model can be used for debugging as well as for computing distinguishing test cases. In the section afterwards, we introduce the diagnosis algorithm using constraints and mutations. This section is followed by the presentation and discussion of the obtained empirical results. Finally, we discuss related research and conclude the paper.

2 BASIC DEFINITIONS

In order to be self contained we briefly introduce the basic definitions. This includes the definition of a test case and test suites, the debugging problem, as well as the definition of mutations. The paper deals with debugging based on models of programs, which are written in a programming language. In this paper we assume an imperative, sequential assignment language $\mathcal{L}$ with syntax and semantics similar to Java ignoring all object-oriented constructs and method calls. We further restrict the data domain of the language to integers and booleans. In Figure 1 we state an example program, which serves as running example. The program implements the division of two integer numbers where a bug is introduced in Line 1.

In order to state the debugging problem, we assume a program $\Pi \in \mathcal{L}$ that does not behave as expected. In the context of this paper such a program Π is faulty when there exist input values from which the program computes output values differing from the expected values. The input and correct output values are provided to the program by means of a test case. For defining test cases we introduce variable environments (or environments for short). An environment is a set of

pairs (x, v) where x is a variable and v its value. In an environment there is only one pair for a variable. We are now able to define test cases formally as follows:

Definition 1 (Test case) *A test case for a program $\Pi \in \mathcal{L}$ is a tuple (I, O) where I is the input variable environment specifying the values of all input variables used in Π, and O the output variable environment not necessarily specifying values for all output variables.*

For example a (failing) test case for the program from Fig. 1 is $I_\Pi : \{a = 2; b = 1\}$ and $O_\Pi : \{result = 2\}$.

A test case is a *failing test case* if and only if the output environment computed from the program Π when executed on input I is not consistent with the expected environment O, i.e., when $\textbf{exec}(\Pi, I) \not\supseteq O$. Otherwise, we say that the test case is a *passing test case*. If a test case is a failing (passing) test case, we also say that the program fails (passes) executing the test case. For the program from Fig. 1 the test case (I_Π, O_Π) is a failing test case. For input I_Π the program will return $result = 3$ which contradicts the expected output $O_\Pi : \{result = 2\}$.

Definition 2 (Test suite) *A test suite TS for a program $\Pi \in \mathcal{L}$ is a set of test cases of Π.*

A program is said to be correct with respect to TS if and only if the program passes all test cases. Otherwise, we say that the program is incorrect or faulty. This definition of correctness is similar to the input output conformance relation (IOCO) from Tretmans (Tretmans, 1996).

We are now able to state the debugging problem.

Definition 3 (Debugging problem) *Let $\Pi \in \mathcal{L}$ be a program and TS its test suite. If $T \in TS$ is a failing test case of Π, then (Π, T) is a debugging problem.*

A solution to the debugging problem is the identification and correction of a part of the program responsible for the detected misbehavior. We call such a program part an explanation. There are many approaches that are capable of returning explanations including (Mayer *et al.*, 2009; Weimer *et al.*, 2009; Mayer, 2007) and (Ceballos *et al.*, 2006; Nica *et al.*, 2009) among others. In this paper, we follow the debugging approach based on constraints, i.e., (Ceballos *et al.*, 2006; Nica *et al.*, 2009). In particular, the approach makes use of the program's constraint representation to compute possible fault candidates. So, debugging is reduced to solving the corresponding constraint satisfaction problem (CSP).

Definition 4 (Constraint Satisfaction Problem (CSP)) *A constraint satisfaction problem is a tuple (V, D, CO) where V is a set of variables defined over a set of domains D connected to each other by a set of arithmetic and boolean relations, called constraints CO. A solution for a CSP represents a valid instantiation of the variables V with values from D such that none of the constraints from CO is violated.*

Note that the variables used in a CSP are not necessarily variables used in a program. We discuss the representation of programs as a CSP in the next section. Afterwards we introduce an algorithm for computing bug candidates from debugging problems. This algorithm only states statements as potential explanations for a failing test cases. No information regarding how to correct the program is given. Hence, we have to extend the approach to deliver also repair suggestions. This is done by mutating program fragments.

In the context of our paper we define program mutation as follows.

Definition 5 (Mutant) *Given a program Π and a statement $S_\Pi \in \Pi$. Further let S'_Π be a statement that results from S_Π when applying changes like modifying the operator or a variable. We call the program Π', which we obtain when replacing S_Π with S'_Π, the mutant of program Π with respect to statement S_Π.*

Another important issue in the theory of program mutation is the identification of a test case able to outline the semantical difference between a program and its mutant. We call such a test case a *distinguishing test case*.

Definition 6 (Distinguishing test case) *Given a program $\Pi \in \mathcal{L}$ and one of its mutant Π', a distinguishing test case for program Π and its mutant Π' is a tuple $(I, \emptyset)$ such that for the input value I the output value of program Π differs from the output value of program Π'.*

In the next section we discuss the conversion of programs into their corresponding constraint representation.

3 CSP REPRESENTATION OF PROGRAMS

Before converting a program $\Pi \in \mathcal{L}$ into its corresponding constraint representation we have to apply some intermediate transformation steps. These transformations are necessary for removing its imperative behavior, i.e., making it a declarative one, as required by the constraint programming paradigm.

Our three step algorithm for converting a program and encoding its debugging problem into a CSP, is as follows:

1. *Loop elimination $\Pi_{LF} = LR(\Pi)$:* We define loop-elimination as a recursive function where n is the number of iterations:

$$LF(\text{while } C\ \{B\}, n) = \begin{cases} \text{if } C\{B\ LF(\text{while } C\ \{B\}, n-1)\} & \text{if } n = 0 \\ \epsilon & \text{otherwise} \end{cases}$$

   We replace each loop-structure by a number of nested if-statements, i.e., number of iterations. The number of iterations n, is given by the test case. The two-iterations version of the program from Figure 1 is given in Figure 2.

2. *SSA conversion $\Pi_{SSA} = SSA(\Pi_{LF})$:* The static single assignment (SSA) form is an intermediate representation of a program with the property that no two left-side variable share the same name. This property of the SSA form allows for an easy conversion into a CSP. It is beyond our scope to detail the program-to-SSA conversion. However, to be self-contained we only explain the necessary rules needed for converting our running example into its SSA representation. For more details regarding the SSA-conversion see for example (Wotawa and Nica, 2008).

```
1.     tmp = (a + 1); // ERROR
2.     if (b == 0) {
3.         result = -1;
        } else {
4.         result = 0;
5.         if (tmp > 0) {   // first iteration
6.             result = result + 1;
7.             tmp = tmp - b;
8.             if (tmp > 0) { //second iteration
9.                 result = result + 1;
10.                tmp = tmp - b;
                }}}
```

Figure 2: Two iteration unrolling for the program from Figure 1

```
1.  tmp_1=(a_0+1);
2.  cond_0=b_0==0;
3.  result_1=-1;
4.  result_2=0;
5.  cond_1=(!cond_0 ∧ tmp_1>0);
6.  result_3=result_2+1;
7.  tmp_2=tmp_1-b_0;
8.  cond_2=(cond_1 ∧ tmp_2>0);
9.  result_4=(result_3+1);
10.  tmp_3=tmp_2-b_0;
11.  result_5=Φ(result_3,result_4,cond_2);
12.  tmp_4=Φ(tmp_2,tmp_3,cond_2);
13.  result_6=φ (result_2,result_5,cond_1);
14.  tmp_5=Φ(tmp_1,tmp_4,cond_1);
15.  result_7=Φ(result_6,result_1,cond_0);
16.  tmp_6=Φ(tmp_5,tmp_1,cond_0);
```

Figure 3: The SSA form corresponding to the program from Figure 2

- We convert ***assignments*** by adding an index to a variable each time the variable is defined, i.e., occurs at the left side of an assignment. If a variable is re-defined, we increase its unique index by one such that the SSA-form property holds. The index of a referenced variable, i.e., a variable occurring at the right side of an assignment, equals to the index of the last definition of the variable.
- We split the conversion of ***conditional structures*** into three steps: (1) the entry condition is saved in an auxiliary variable, (2) each assignment statement is converted following the above rule, and (3) for each conditional statement and variable defined in the sub-block of the statement, we introduce an evaluation function

  $$\Phi(\mathtt{v_{then}}, \mathtt{v_{else}}, \mathtt{cond}) \stackrel{\text{def}}{=} \begin{cases} \mathtt{v_{then}} & \text{if } \mathtt{cond} = true \\ \mathtt{v_{else}} & \text{otherwise} \end{cases}$$

  which returns the statement conditional-exit value, e.g., $\mathtt{v_{after}} = \Phi(\mathtt{v_{then}}, \mathtt{v_{else}}, \mathtt{cond})$.

The SSA representation of the program from Figure 2 is given in Figure 3.

3. *Constraint conversion* $CON = CC(\Pi_{SSA})$*:* This last step of the conversion process additionally to converting the SSA statements to the corresponding constraints, also includes the encoding of the debugging problem. For this purpose we introduce a special boolean variable $AB(S)$ for a statement S, that states the incorrectness of a statement S. The constraint model of a statement comprises corresponding constraints or-connected with $AB(S)$. Let $S \in \Pi_{SSA}$ and let C_S be the constraint encoding statement S in the constraint programing language. Note that ϕ functions cannot be incorrect. Hence, no AB variable is defined for statements using ϕ. We model S in CON as follows:

   $$CON \cup \begin{cases} \mathtt{AB(S)} \vee \mathtt{C_S} & \text{if } S \text{ does not contain } \phi \\ \mathtt{C_S} & \text{otherwise} \end{cases}$$

   Hence the CSP representation of a program Π is given by the tuple $(V_{\pi_{SSA}}, D_{SSA}, CON)$, where $V_{\pi_{SSA}}$ represents all variables of the SSA representation Π_{SSA} of program Π, defined over the domains $D_{SSA} = \{Integer, boolean\}$.

   Debugging of a program requires the existence of a failing test case. This means that, in addition to the set of constraints CON, we must add an extra set of constraint encoding a failing test case (I, O). For all $(x, v) \in I$ the constraint $x\_0 = v$ is added to the constraint system. For all $(y, w) \in O$ the constraint $y\_\iota = w$ is added where ι is the greatest index of variable y in the SSA form. Let CON_{TC} denote the constraints resulted from converting the given test case. Then, the CSP corresponding to the debugging problem of a program Π is now represented by the tuple $(V_{\pi_{SSA}}, D_{SSA}, CON \cup CON_{TC})$

   In our implementation we model the CSP to represent the debugging problem in the language of the MINION constraint solver (Gent *et al.*, 2006). MINION is an out of the box, open source constraint solver. Its syntax requires little effort in modeling the constraints than other constraint solvers, e.g., it does not support different operators on the same constraint. Because of this drawback sometimes complex constraints have to be split into two or three more simpler constraints. However, because of this characteristic, MINION, unlike other constraint solver toolkits, does not have to perform an intermediate transformation of the input constraint system.

After explaining the conversion of debugging problems into CSP, in the following section we discuss the debugging algorithm and its extension with mutations and distinguishing test cases.

4 DEBUGGING

The debugging approach presented in the paper comprises 3 steps. The first step comprises the computation of bug candidates, i.e., program statement that might cause the revealed misbehavior, from the constraint representation of a program $\Pi \in \mathcal{L}$. In the second step, for each candidate a set of mutants is computed that would lead to a new program passing all previously failing test cases. If no such mutant can be found the bug candidate is removed from the list of potential candidates. In the third step, distinguishing test cases are computed that allow choice between two randomly selected bug candidates. The third step can be executed several times to further reduce the number of bug candidates. In this section, we explain each of the debugging steps starting from the computation of candidates using the CSP representation to the computation and use of distinguishing test cases.

Let CON_Π be the constraint representation of a program Π and CON_T the constraint representation of a failing test case T. The debugging problem formulated as a CSP comprises CON_Π together with CON_T. Note that in CON_Π assumptions about correctness or incorrectness of statements are given, which are represented by a variable AB assigned to each statement. The algorithm for computing bug candidates calls the CSP solver using the constraints and asks for a return value of AB as a solution. The size of the solution corresponds to the size of the bug, i.e., the number of statements that must be changed together in order to explain the misbehavior. We assume that single statement bugs are more likely than bugs comprising more statements. Hence, we ask the constraint solver for smaller solutions first. If no solution of a particular size is found, the algorithm increases the size of the solutions to be searched for and iterates calling the constraint solver. This is done until either a solution is found or the maximum size of a bug, which is equivalent to the number of statements in Π, is reached.

Algorithm CSP_Debugging (CON_Π, CON_T)
Inputs: A constraint representation CON_Π of a program Π, and a constraint representation CON_T of a failing test case T.
Outputs: A set of minimal bug candidates.

1. Let i be 1.
2. While i smaller or equal to the number of statements in Π do:
   (a) Call the constraint solver using CON_Π, CON_T to search for solutions regarding the AB variables, where only i statements are allowed simultaneously to be incorrect.
   (b) If the constraint solver returns a non-empty set of solution, then return this set as result and leave the algorithm.
   (c) Otherwise, let i be $i + 1$.
3. Return the empty set as result.

For example, for the constraint system corresponding to the program from Fig. 3 the constraint solver MINION finds 5 possible explanations for the failing test case $I : (a\_0 = 0, b\_0 = -250), O : (result\_7 = 0)$ in less the 0.1s. This result is very satisfactory, especially with respect to computation time. However, further steps might be performed in order to reduce the size of the bug candidates. For this purpose we suggest to use mutations.

Assume a faulty program Π and a failing test case (I, O). Let D_{AB} be the set of bug candidates obtained when calling **CSP_Debugging** on the constraint representation of Π and (I, O). The following algorithm makes use of program mutations for further restricting D_{AB}.

Algorithm Filter_TestCase (D_{AB}, Π, T)
Inputs: A set of bug candidates D_{AB}, the faulty program Π, and the failing test case T.
Outputs: A set of mutants Mut_Π of program Π.

1. Let Mut_Π be the empty set.
2. For all elements $d \in D_{AB}$ do:
   (a) Generate all mutants of program Π with respect to the statements stored in d and store them in V_{Mut}.
   (b) Add every program $\Pi' \in V_{Mut}$ passing test case T to Mut_Π.
3. Return Mut_Π.

The **Filter_TestCase** algorithm returns for the faulty program Π a set of repair possibilities Mut_Π. Due to the usage of the debugging algorithm **CSP_Debugging**, we compute the repair only for the resulting bug candidates set D_{AB}. A mutant is part of Mut_Π, i.e., a repair, if and only if it is able to pass the failing test case T. Hence, we expect that the number of bug candidates can be reduced. Moreover, since mutation is only applied for bug candidates we do not need to compute all possible mutations even in the case when they cannot explain the revealed misbehavior.

The number of repair possibilities for a statement of the D_{AB} set is strongly tied to the capabilities of the used mutation operators and the used mutation tool. Because of this fact this part of the approach is as good as the available capability of the used mutation tool. Note that after applying the **Filter_TestCase** algorithm, in our experiments we were able to eliminate between 20% and 60% of the bug candidates, because of the inability of the suggested repair to pass the test case. Hence, filtering based on mutations was very successful.

The last step of our algorithm comprises the integration of distinguishing test cases to further reduce the bug candidate set. Let Mut_Π be the set of mutants for a program Π obtained after applying the **Filter_TestCase** algorithm. And let CON_{Mut_Π} be the constraint representation of the programs from Mut_Π.

Algorithm TestCase_Generator Mut_Π, CON_{Mut_Π}
Inputs: A set of valid repair possibilities, Mut_Π, for a faulty program Π and their constraint representation CON_{Mut_Π}.
Outputs: A subset of Mut_Π.

1. Let $Tested$ be empty.
2. If there exists mutants $\Pi', \Pi'' \in Mut_\Pi$ with $(\Pi', \Pi'') \notin Tested$, add (Π', Π'') to $Tested$ and proceed with the algorithm. Otherwise, return Mut_Π
3. Let $CON_{\Pi'}$ and $CON_{\Pi''} \in CON_{Mut_\Pi}$ be the constraint representation of programs Π' and Π'' respectively.
4. Let CON_{TC} be the constraints encoding $Input_{\Pi'} = Input_{\Pi''} = I \wedge Output_{\Pi'} \neq Output_{\Pi''}$
5. Solve the CSP: $CON_{\Pi'} \cup CON_{\Pi''} \cup CON_{TC}$ using a constraint solver.
6. Let O be the correct output for the original program Π on input I (derived from user interaction or specifications).
7. If $Output_{\Pi'} = O \wedge Output_{\Pi''} \neq O$, then delete Π'' from Mut_Π.
8. If $Output_{\Pi''} = O \wedge Output_{\Pi'} \neq O$, then delete Π' from Mut_Π.
9. If $Output_{\Pi'} \neq O \wedge Output_{\Pi''} \neq O$, delete Π' and Π'' from Mut_Π.
10. If (CSP has no solution) go to step 1.
11. For all $\Pi' \in Mut_\Pi$ do:

(a) If Π' fails on generated test case (I, O) delete Π' from Mut_{Π}.

12. Return Mut_{Π}.

The above algorithm searches for two mutants, distinguished via a test case. The algorithm in the current form is restricted to search for only one pair of such mutants but can be easily changed in order to compute several different pairs where a distinguishing test case is available. The only disadvantage of this algorithm is that Step 6 requires an interaction with an oracle. If no automated oracle is available user interactions are required and prevent the approach from being completely automated. To solve the constraint system resulted at step 5 we use the MINION constraint solver. Another particularity of this approach is that, for the CSP to be solvable, the name of the variables of the two mutants should differ. This is however an encoding problem which can be easily overcome by encapsulating in the name of each variable the name of the mutant file. When using the above approach for the example from 2 we are able to reduce the conflict set to one element, which was also the correct one. For more information regarding distinguishing test cases and their computation using MINION, we refer the interested reader to (Wotawa *et al.*, 2010).

To obtain the program's set of mutants relative to the set of fault candidates we rely on the JAVA mutation tool MuJava (Yu-Seung Ma and Kwon., 2005). MuJava is a Java based mutation tool, which was originally developed by Offut, Ma, and Kwon. Its main three characteristics are:

1. Generation of mutants for a given program.
2. Analysis of the generated mutants.
3. Running of provided test cases.

Due to the new implemented add-ons, the tool supports a command line version for the mutation analysis framework, which offers an easy integration of the tool in the testing or debugging process.

Offutt proved that the computational cost for generating and executing a large number of mutants can be expensive, and thus he proposed a selective mutation operator set that is used by the MuJava tool. It works with both types of mutation operators:

- Method level mutation operators (also called traditional), which modify the statements inside the body of a method;
- Class level mutation operators, which try to simulate faults specific to the object oriented paradigm (for example faults regarding the inheritance or polymorphism).

For our experiments we take into account only the traditional mutation operators. Moreover, we further restrict the mutation operators to mutations on expressions comprising deletion, replacement, and insertion of primitive operators (arithmetic operators, relational operators, conditional operators, etc.). Mutation by deletion of operands or statements was proved to be inefficient (A. J. Offutt and Zapf, 1996). Because of the selected tools there are currently some limitations of our implementation. If the bug is on the left side of an assignment we cannot correct it. Another limitation is with respect to constants. If the bug is due to an initialization, MuJava is not able to generate any mutants. Missing statements are another limitation of the approach. We currently do not consider bugs because of missing statements. Finally, there is a limitation regarding multiple bugs in one statement. In this case the MuJava tool is not able to mutate more than one variable or operator per statement and mutant, i.e., each mutant contains only one change when compared with the original program.

5 EMPIRICAL RESULTS

We tested our approach against a set of faulty programs. In each program we manually injected one single fault. All the faults are found at the right side of the assignment and with the exception of the *tcsa03* program all faults are functional faults. We used as test oracle the original bug free version of each faulty program. Using the output values of the original bug-free program we were able to decide which of the mutants are to be eliminated after computing the distinguishing test cases. In the real life situation we cannot benefit from the existence of such a program. Therefore, we must rely on the user or a given formal specification to determine the correct output for a given input.

The process of mutant generation, program to CSP conversion, and the computation of the conflict set is fully automated. However the generation of the distinguishing test cases was performed manually.

In order to obtain the empirical results, we applied the following process. For each program we first performed the conversion into its constraint representation. Then we computed the fault candidates. For each fault candidate, i.e., faulty statement, we computed all its possible mutants. We eliminated from the generated set of mutants all mutants which were not able to pass the error revealing test case. In addition, we tested the number of oracle-interactions required to obtain the minimal set of faulty components. By an oracle-interaction we understand repeating the **TestCase_Generator** algorithm until no other distinguishing test case can be generated, i.e., each time we applied the algorithm we asked the oracle, i.e., the original fault free program in our case, to provide the correct output for the generated test case.

The results of the empirical study are given in Table 1. In most of the cases we were able to eliminate more than half of the initial fault candidates set. Reducing the diagnosis candidates by eliminating those candidates where no mutant that passes the original test suite can be found, is very effective. The use of distinguishing test cases further reduces the number of fault candidates. Thus finally, only one diagnosis candidate remains, which was always the correct one. When using larger programs like *tcas* a reduction to one diagnosis candidate was not possible. However, even in this case the approach lead to a reduction of more than 60 percent regarding the computed diagnosis candidates.

Another factor, which influences the quality of the obtained results, is the way of choosing the mutant pairs for computing distinguishing test cases. There is no way to predict if a certain pair of mutants will produce the best or worst distinguishing test case. Therefore, we randomly selected the pair of mutants when carrying out the empirical evaluation. For example, we observed that after trying out all mutant pairs for the

DivATC_V4 program the best distinguishing test case would lead to 1 element in the conflict set contrary to 3 as given in Table 1.

It is also worth noting that computing the diagnosis candidates and the distinguishing test cases using the CSP solver MINION was very fast. For all examples, the necessary time never exceeded 0.3 seconds using a Pentium 4 Dual core 2 GHz with 4 GB of RAM computer. Hence, for smaller programs or program parts that can be separately analyzed like methods, the proposed approach is feasible.

6 RELATED RESEARCH

Our work is mainly based on model-based diagnosis (Reiter, 1987) and its application to debugging (Mayer, 2007; Mayer and Stumptner, 2003). In contrast to previous work we are not using logic-based models of programs but a constraint representation and a general constraint solver. The most similar work in this respect is (Ceballos *et al.*, 2006; Nica *et al.*, 2009; Wotawa and Nica, 2008). Instead of focusing only on constraint-based debugging, we combine fault localization with mutations and testing.

In (Weimer *et al.*, 2009) and more recently (Debroy and Wong, 2010) the authors describe the application of mutations and genetics programming to software debugging. In order to avoid computing too many mutants the authors use focusing techniques based on dependencies and spectrum-based methods respectively. The use of mutations is similar to our work. The difference is that we are using constraint-based debugging for focusing and integration of testing for reducing the size of the conflict set, which, to the best of our knowledge, has not been introduced before.

Other more recent approaches of debugging include delta debugging (Zeller and Hildebrandt, 2002), spectrum-based debugging (Jones and Harrold, 2005; Abreu *et al.*, 2009), and slicing based methods like (Kusumoto *et al.*, 2002; Binkley and Harman, 2004; Zhang *et al.*, 2005). The focus of our approach is on generating automated tests for distinguishing diagnosis candidates and thus to further make automated debugging more accessible and useful in practice.

7 CONCLUSION

In this paper we presented an approach for restricting the number of potential diagnosis candidates by providing distinguishing test cases. A distinguishing test case for two diagnosis candidates is characterized by a set of inputs that reveal different executions for both diagnosis candidates such that they can be distinguished with respect to their output behavior. Just using the distinguishing test case alone we are not able to decide which diagnosis candidates to remove or if we should eliminate both from the list of candidates. This can only be done after consulting a test oracle, e.g., the user or a formal specification, for the expected output of the distinguishing test case. Candidates where the computed output is not equivalent to the expected one can be eliminated. The advantage of this approach is that only the input-output behavior of a program is used for distinguishing diagnosis candidates. Moreover, the approach computes additional test cases based on their discriminating power for distinguishing diagnosis candidates. Usually, test cases

Name	It	$\mathbf{Var}_{\Pi}$	$\mathbf{LOC}_{\Pi}$	Inputs	Outputs	$\mathbf{LOC}_{\mathbf{SSA}}$	$\mathbf{CO}$	$\mathbf{Var}_{\mathbf{CO}}$	$\mathbf{Diag}$	$\mathbf{Diag}_{\mathbf{filt}}$	#UI	$\mathbf{Diag}_{\mathbf{TC}}$
DivATC_V1	2	5	21	2	1	32	33	29	3	2	1	2
DivATC_V2	2	5	21	2	1	32	33	29	5	3	1	1
DivATC_V3	2	5	21	2	1	32	33	29	3	2	1	2
DivATC_V4	2	5	21	2	1	32	33	29	4	4	1/2	3(1)/1
GcdATC_V1	2	6	35	2	1	49	61	46	2	2	1	1
GcdATC_V2	2	6	35	2	1	49	61	46	10	3	1/2/3/4/5	3/3/2/2/1
GcdATC_V3	2	6	35	2	1	49	61	46	2	2	1	1
MultATC_V1	2	5	16	2	1	26	24	19	2	2	1	1
MultATC_V2	2	5	16	2	1	26	24	19	2	2	1	1
MultATC_V3	2	5	16	2	1	26	24	19	2	2	1	1
MultATC_V4	2	5	16	2	1	26	24	19	5	2	1	1
MultV2ATC_V1	2	6	20	2	1	49	67	46	6	2	1	1
MultV2ATC_V2	2	6	20	2	1	49	67	46	2	1	1	1
MultV2ATC_V3	2	6	20	2	1	49	67	46	6	1	1	1
SumATC_V1	2	5	18	2	1	27	24	20	2	2	1	1
SumATC_V2	2	5	18	2	1	27	24	20	3	2	1	1
SumATC_V3	2	5	18	2	1	27	24	20	5	2	1	1
SumPowers_V1	2	11	36	3	1	72	87	70	16	6	1/2/3/4	4/4/2/2
SumPowers_V2	2	11	36	3	1	72	87	70	11	6	1/2	2/1
SumPowers_V3	2	11	36	3	1	72	87	70	11	1	1	1
tcas08	1	48	125	12	1	125	98	132	27	13	1/2/3/4	11/11/11/10
tcas03	1	48	125	12	1	125	98	132	27	13	1/2/3/4	13/12/9/9

Table 1: Each program **Name**, has associated a number of iterations **It**, the number of variables $\mathbf{Var}_{\pi}$, its size given in lines of code $\mathbf{LOC}_{\mathbf{\Pi}}$, the number of inputs **Inputs**, number of outputs **Outputs**, the size of its SSA representation given as lines of code$\mathbf{LOC}_{\mathbf{SSA}}$, the number of MINION constraints $|\mathbf{CO}|$, the number of MINION variables over which the constraint system is defined $\mathbf{Var}_{\mathbf{CO}}$, the number of fault candidates $|\mathbf{Diag}|$, the size of the conflict set resulted after applying $Filter\_TestCase$ algorithm, $|\mathbf{Diag}_{\mathbf{filt}}|$, the number of calls to the $TestCase_Generator$ algorithm, **#UI** to obtain the number of fault candidates $|\mathbf{Diag}_{\mathbf{TC}}|$.

are generated for fulfilling coverage criteria like statement coverage or branch coverage.

Beside the theoretical contribution we present first empirical results of the proposed approach. The results indicate that the approach allows a substantial reduction of the diagnosis candidates. For smaller programs we were able to reduce the diagnosis candidates to the real bug. Obviously, this was not always the case. For larger programs more diagnosis candidates remain. This has been somehow expected because programs cannot be usually corrected only by replacing one statement with another. Instead the right repair actions might comprise changes at different positions in the program. In future work we want to extend the empirical study. This includes to use more and larger programs as well as example programs comprising multiple faults.

REFERENCES

(A. J. Offutt and Zapf, 1996) G. Rothermel R. Untch A. J. Offutt, A. Lee and C. Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology*, 5:99–118, 1996.

(Abreu *et al.*, 2009) Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. Spectrum-based multiple fault localization. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 88–99, 2009.

(Binkley and Harman, 2004) David Binkley and Mark Harman. A survey of empirical results on program slicing. In Marvin Zelkowitz, editor, *Advances in Software Engineering – Advances in Computers Vol. 62*, pages 106–172. Academic Press Inc., 2004. See also citeseer.ist.psu.edu/661032.html.

(Ceballos *et al.*, 2006) R. Ceballos, R. M. Gasca, C. Del Valle, and D. Borrego. Diagnosing errors in dbc programs using constraint programming. *Lecture Notes in Computer Science*, 4177:200–210, 2006.

(Debroy and Wong, 2010) Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Third International Conference on Software Testing, Verification and Validation (ICST 2010)*. IEEE, 2010.

(Gent *et al.*, 2006) I. P. Gent, C. Jefferson, and I. Miguel. Minion: A fast, scalable, constraint solver. *17th European Conference on Artificial Intelligence*, ECAI-06, 2006.

(Jones and Harrold, 2005) J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings ASE'05*, pages 273–282. ACM Press, 2005.

(Kusumoto *et al.*, 2002) Shinji Kusumoto, Akira Nishimatsu, Keisuke Nishie, and Katsuro Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, 7:49–76, 2002.

(Mayer and Stumptner, 2003) W. Mayer and M. Stumptner. Model-based debugging using multiple abstract models. *Proceedings of the 5th International Workshop on Automated and Algorithmic Debugging*, AADEBUG-03:55–70, 2003.

(Mayer *et al.*, 2009) Wolfgang Mayer, Rui Abreu, Markus Stumptner, and Arjan J.C. van Gemund. Prioritising model-based debugging diagnostic reports. In *Proceedings of the International Workshop on Principles of Diagnosis (DX)*. 2009.

(Mayer, 2007) W. Mayer. Static and hybrid analysis in model-based debugging. *PhD Thesis, School of Computer and Information Science*, University of South Australia, 2007.

(Nica *et al.*, 2009) M. Nica, J. Weber, and F. Wotawa. On the use of specification knowledge in program debugging. *20th International Workshop on Principles of Diagnosis*, (DX-09), 2009.

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

(Tretmans, 1996) J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

(Weimer *et al.*, 2009) Westley Weimer, Thanh Vu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 512–521, 2009.

(Wotawa and Nica, 2008) F. Wotawa and M. Nica. On the compilation of programs into their equivalent constraint representation. *Informatika*, 32:359–371, 2008.

(Wotawa *et al.*, 2010) Franz Wotawa, Mihai Nica, and Bernhard K. Aichernig. Generating distinguishing tests using the minion constraint solver. In *CSTVA 2010: Proceedings of the 2nd Workshop on Constraints for Testing, Verification and Analysis*. IEEE, 2010.

(Yu-Seung Ma and Kwon., 2005) Jeff Offutt Yu-Seung Ma and Yong Rae Kwon. Mujava : An automated class mutation system. *Software Testing, Verification and Reliability*, 15:97–133, 2005.

(Zeller and Hildebrandt, 2002) Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2), feb 2002.

(Zhang *et al.*, 2005) Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. Experimental evaluation of using dynamic slices for fault localization. In *Sixth International Symposium on Automated & Analysis-Driven Debugging (AADEBUG)*, pages 33–42, 2005.