Generating Distinguishing Tests using the MINION Constraint Solver

Franz Wotawa and Mihai Nica and Bernhard K. Aichernig
Institute for Software Technology
Technische Universität Graz, Graz, Austria
Email: {wotawa,nica,baichern}@ist.tugraz.at

Abstract—We discuss the generation of test cases for demonstrating the non-equivalence of two programs. This problem has applications in mutation testing and automated debugging. After transforming the programs into Static-Single-Assignment form, the MINION constraint solver is used to generate test vectors that demonstrate the observable difference. The experiments show the feasibility of our constraint solving approach.

I. INTRODUCTION

Constraints have been used for various purposes like verification [4], debugging [2],[13], program understanding [14] as well as testing [6],[9],[10]. Some of the proposed techniques use constraints to state specification knowledge like pre- and post-conditions. Others use constraints for modeling purposes or extract the constraints directly from the source code. In this paper we rely on the latter and use constraints obtained from the program directly. In contrast to previous research we focus on generating test cases that can be used to distinguishing different implementations. A test case distinguishes two implementations if it reveals a different output behavior using the same inputs for both implementations. Of course such a distinguishing test case might not exists always. Moreover, we assume that the implementations behave deterministically. Otherwise, it is not guaranteed that a given input always generates the same outputs.

There are many potential applications of distinguishing test cases. The first application scenario is test case generation based on program mutations. In such an application mutants for a given program are generated. The distinguishing test case generator is used to compute test cases for each mutant and the original program. The other scenario is debugging. In debugging we might obtain too many diagnosis candidates, i.e., parts of the program that explain a detected misbehavior. In order to reduce the diagnosis candidates we have to add new information like test cases. The distinguishing test case generator delivers this new test cases.

The idea behind our approach is to convert two implementations into constraints and to represent the problem of generating distinguishing test cases as constraint satisfaction problem. In order to convert a program into a constraint system we first remove the loops and iterations by replacing them basically with a bounded sequence of conditionals. Then we compile the resulting program into its static single assignment form from which we are able to compute the constraints directly. For the constraint representation and the solving we rely on the MINION constraint solver [8],[11].

Before discussing our technique for generating distinguishing test cases in detail, we outline the underlying ideas on a small example program. More details are given in the rest of this paper.

1. begin
2. \( i = 2 \times x; \)
3. \( j = 2 \times y; \)
4. \( o_1 = i + j; \)
5. \( o_2 = i \times i; \)
6. end;

This program can be easily converted into a constraint representation using the constraint language from MINION. We only need to convert the program statement by statement. For representing constraints we use a relational notation. For example, the multiplication \( x \times y = z \) is represented by \( \text{product}(x,y,z) \) and for the sum \( x+y = z \) we use the two relations \( \text{sumleq}([x,y],z) \) and \( \text{sumgeq}([x,y],z) \) stating \( x+y \leq z \) and \( x+y \geq z \) respectively. Hence, the constraint representation of our program is the following:

\[
\begin{align*}
\text{product}(2,x,i) \\
\text{product}(2,y,j) \\
\text{sumleq}([i,j],o1) \\
\text{sumgeq}([i,j],o1) \\
\text{product}(i,i,o2)
\end{align*}
\]

Now consider a variant of the program where Line 3 is changed to \( j = 3 \times y \) and let us again convert it into its constraint representation. Note that in this case we added a post-fix string "\_v" to each variable to distinguish the variables of the original program from the variables of the variant.

\[
\begin{align*}
\text{product}(2,x_v,i_v) \\
\text{product}(3,y_v,j_v) \\
\text{sumleq}([i_v,j_v],o1_v) \\
\text{sumgeq}([i_v,j_v],o1_v) \\
\text{product}(i_v,i_v,o2_v)
\end{align*}
\]

Informally speaking a distinguishing test case is a test case for separating the behavior of two programs where the input values for each program is the same but the computed output is not. Hence, we have to state that the inputs are the same and that there exists at least one output where the computed values are not equivalent. Using the MINION constraint eq for stating equivalence, diseq for stating that two variables are inequivalent, and the logical constraint watched-or for
formalizing a disjunction, we give the constraints necessary to obtain a distinguishing test case in our example:

\[
\begin{align*}
\text{eq}(x, x_v) \\
\text{eq}(y, y_v) \\
\text{watched-or}(\{\text{diseq}(01, 01_v), \text{diseq}(02, 02_v)\})
\end{align*}
\]

A solution for the given constraints is also a distinguishing test case. Using MINION as constraint solver we are able to compute more than one distinguishing test cases for this example, e.g., one solution is \(x=2, y=2\) and there are many others. All solutions have in common that \(y\) is not equal to 0.

The rest of this paper is organized as follows. In the next section we discuss the program conversion step and give all necessary basic definitions. Afterwards, we outline the algorithm for computing distinguishing test cases and present first empirical results. Finally, we discuss related research and conclude the paper.

II. BASIC DEFINITIONS

We assume that the program \(\Pi\) to be compiled into a constraint representation is deterministic and written in an imperative assignment language with the usual kinds of statements, e.g., variable assignments, conditional statements, and loops. The underlying type system of the language comprises basic datatypes like Booleans, integers, floating point numbers and arrays. Each variable stores a value of the corresponding datatype. The values of variables are stored in a variable environment. A variable environment (or environment for short) is a function mapping variables to their values. We further assume that each program \(\Pi\) has some input variables and output variables. We use \([\Pi](I)\) to denote execution of \(\Pi\) on a specific input environment (or input for short) \(I\). The result of the execution is always an environment, i.e., the output environment. In the following we also represent environments as set of tuples \((x, v)\) where \(x\) is a variable and \(v\) is a value.

A test case is a tuple \((I, O)\) where \(I\) is the input environment denoting the given values of input variables, and \(O\) is the output environment where the expected values of the output variables are specified. Note that regarding the definition it is also possible that \(O\) is empty. A program \(\Pi\) is passing a test case \((I, O)\) if and only if the execution of \(\Pi\) on \(I\) returns the expected output values specified in \(O\). Formally, we define passing and failing as follows:

\[\[\Pi\](I) \supseteq O \iff \Pi\text{ passes test case } (I, O)\]

\[\neg (\Pi\text{ passes test case } (I, O)) \iff \Pi\text{ fails test case } (I, O)\]

Note that not all values have to be specified. However, it is necessary that all given values are returned as expected. A variable where no value is specified in \(O\) can have an arbitrary value after program execution.

**Definition 1 (Distinguishing test case):** Given programs \(\Pi\) and \(\Pi'\). A test case \((I, \emptyset)\) is a distinguishing test case if and only if there is at least one output variable where the value computed when executing \(\Pi\) is different from the value computed when executing \(\Pi'\) on the same input \(I\).

\((I, \emptyset)\) is distinguishing \(\Pi\) from \(\Pi'\) \(\iff\)

\[\exists x: (x, v) \in [\Pi](I) \land (x, v') \in [\Pi'](I) \land v \neq v'\]

We call the problem of finding an input environment that distinguishes two programs \(\Pi\) and \(\Pi'\) the **distinguishing test case problem**. It is worth noting that a distinguishing test case is according to our definitions always a passing test case because the output environment is not specified. From the distinguishing test case we always are able to derive a test case with a specified expected output. This can be done manually or in some cases automatically. The latter is used to compute test cases from the mutations of a given program. In this case the program is assumed to be correct and the output of the execution of the program is therefore the expected output of the test case. When searching for distinguishing test cases for all mutants we finally receive a test suite that can be used to separate the original program from all its mutations.

A. Conversion of programs to constraints

The problem of computing distinguishing test cases can be reduced to a constraint satisfaction problem (CSP). The only requirement is that we are able to compile programs into an equivalent constraint representation. Based on previous work \([4]\), \([7]\) and to be self contained we briefly discuss this compilation process.

Our work addresses sequential programs with a syntax and semantics similar to well-known languages like Java, but without object-oriented constructs. For simplicity we do not consider procedure calls and arrays in this paper, but it should be noted that procedures and arrays can be straightforwardly integrated as shown in \([13]\). Our approach supports assignment statements, conditional statements, and loops. Figure 1 depicts a program which serves as running example throughout the rest of the paper.

```plaintext
int power(int a, int exp)
1. int e = exp;
2. int res = 1;
3. while (e > 0) {
4.   res = res * a;
5.   e = e - 1;
7. }
6. return res;
```

Fig. 1. A program for computing \(a^\text{exp}\), where \(a\) and \(\text{exp}\) are integers. Variable \(\text{res}\) denotes the result.

As loops cannot be directly converted to our constraint representation, past works have proposed to **unroll** loops, i.e., to create a loop-free program by replacing the loop of the original program by a set of nested if-statements (e.g., see \([2]\) and \([13]\)). If the maximum number of iterations is known in advance, then the loop-free program is equivalent to the original program. Note that in our case we are able to restrict the number of loops searching for distinguishing test cases within the given maximum number of iterations. Figure 2 shows the loop-free version of our example program for 2 iterations.
int power_loopfree(int a, int exp)
1. int e = exp;
2. int res = 1;
3. if (e > 0) {
4.    res = res * a;
5.    e = e - 1;
6. } else {
7.    res = res * a;
8.    e = e - 1;
9. }
10. return res;

Fig. 2. The loop-free version of the program in Fig. 1 for 2 iterations.

Our constraint representation requires that all left-side variables in the program have unique names, i.e., each variable should be defined only once. Hence, we use the Static Single Assignment (SSA) form, which is an intermediate representation of the program with the property that no two left-side variables have the same name (see [3], [1], [12]). This is achieved by replacing each left-side variable with a new variable whose name is composed of the name of the original variable plus a unique index as suffix, see Fig. 3.

In order to obtain the SSA form of a program it is also necessary to convert loops and conditional statements. As loops are, in our approach, represented by nested if-statements, we only need to consider the conversion of conditional statements of the form

\[ \text{if}(\text{cond}_{\text{expr}}) \text{ then } \{ \text{...} \} \text{ else } \{ \text{...} \} \]

Note that the notation \(x_{\text{expr}}\) denotes a whole expression rather than a single variable. In brief, this conversion works as follows:

1) The value of the evaluated condition \(\text{cond}_{\text{expr}}\) is stored in a new boolean variable \(\text{cond}_i\), where \(i\) is a unique index.
2) The if- and the else-branches are converted separately. For both branches, new variables with unique indexes are introduced. The statements of both branches are concatenated; i.e., the program in SSA form will execute the statements of either branch in every run.
3) New variables are added which have the value of the corresponding variables in the original program after the execution of the if- or else-branch, respectively. The values of these new variables depend on the indexed variables which were introduced for the branches and on the boolean condition \(\text{cond}_{\text{expr}}\). For the evaluation of those values we define the \(\Phi\)-function:

\[
\Phi(v_j,v_k,\text{cond}_i) = \begin{cases} v_j & \text{if } \text{cond}_i = \text{true} \\ v_k & \text{otherwise} \end{cases}
\]

For example, the corresponding SSA form of the program fragment

\[ \text{if}(\text{cond}_{\text{expr}}) \{ x = E^1_{\text{expr}}; \} \text{ else } \{ x = E^2_{\text{expr}}; \} \]

is given as follows:

\[ \text{cond}_i = \text{cond}_{\text{expr}}; \]

\[ x = E^1_{\text{expr}}; \]

\[ x = E^2_{\text{expr}}; \]

int power_SSA(int a, int exp)
1. int e_0 = exp;
2. int res_0 = 1;
3. bool cond_0 = (e_0 > 0);
4. int res_1 = res_0 * a;
5. int e_1 = e_0 - 1;
6. bool cond_1 = cond_0 \&\& (e_1 > 0);
7. int res_2 = res_1 * a;
8. int e_2 = e_1 - 1;
9. int res_3 = \Phi(res_2, res_1, \text{cond}_1);
10. int e_3 = \Phi(e_2, e_1, \text{cond}_1);
11. int res_4 = \Phi(res_3, res_0, \text{cond}_0);
12. int e_4 = \Phi(e_3, e_0, \text{cond}_0);

Fig. 3. The loop-free SSA form of the program in Fig. 1 for 2 iterations. Variable \(res_4\) is the output of the program (i.e., the final result).

\[ x_j = E^1_{\text{expr}}; \]
\[ x_k = E^2_{\text{expr}}; \]
\[ x_1 = \Phi(x_j, x_k, \text{cond}_i); \]

The SSA representation for the program in Fig. 1 is depicted in Fig. 3. A noteworthy statement is:

6. bool cond_1 = cond_0 \&\& (e_1 > 0);

The variable \(\text{cond}_1\) is true iff \(e > 0\) holds in the original program after the first loop iteration, i.e., \(\text{cond}_1\) is true iff a second loop iteration is executed. This is the case iff \(e_0 > 0\) and \(e_1 > 0\) holds.

Obviously, the conversion of the loop-free program into the SSA form has, apart from variable renaming, no influence on the actual program behavior. It can also be seen that the SSA form of a sequential program comprises only assignments of the general form

\[ v = E_{\text{expr}} \]

where \(v\) is a variable and \(E_{\text{expr}}\) is either an expression in the syntax of the sequential language or it is an expression of the form \(\Phi(\ldots)\).

What is missing in the conversion process is the mapping of SSA programs to MINION constraints. MINION offers support for almost all arithmetics, relational, and logic operators like minus, plus, multiplication, division, less, and equal over integers, but enforces all expressions used in a MINION program to be limited to one operator.

Because of the syntactical limitations of MINON we have to convert an assignment statement with an expression \(E_{\text{expr}}\) on the right-side comprising more than one operator into a sequence of MINON statements. The idea behind the conversion is simple. A constant or variable is represented by itself. For an expression of the form \(E^1_{\text{expr}} \text{ op } E^2_{\text{expr}}\) we convert \(E^1_{\text{expr}}\) and \(E^2_{\text{expr}}\) separately, and assign a new intermediate variable for each converted sub-expression. The following ComputeExpression algorithm implements the conversion.

Algorithm ComputeExpression\(E_{\text{expr}}\)
Input: An expression \(E_{\text{expr}}\) and an empty set \(M\) for storing the MINION constraints.
Output: A set of minion constraints representing the expression stored in \( M \), and a variable or constant where the result of the conversion is finally stored.

1) If \( \text{expr} \) is a variable or constant, then return \( \text{expr} \).
2) Otherwise, \( \text{expr} \) is of the form \( \text{expr}_1 \oplus \text{expr}_2 \).
3) Let \( \text{aux}_1 = \text{ComputeExpression} (\text{expr}_1) \).
4) Let \( \text{aux}_2 = \text{ComputeExpression} (\text{expr}_2) \).
5) Generate a new MINION variable \( \text{result} \) and create MINION constraints accordingly to the given operator \( \oplus \), which define the relationship between \( \text{aux}_1, \text{aux}_2 \), and \( \text{result} \), and add them to \( M \).
6) Return \( \text{result} \).

For example, the expression \( a_0 + b_0 - c_0 \) is converted to the following MINION constraints using \text{ComputeExpression} where \( \text{aux}_1 \) and \( \text{aux}_2 \) represent new variables introduced during conversion.

\[
\begin{align*}
\text{sumleq}([a_0,b_0], \text{aux}_1) \\
\text{sumgeq}([a_0,b_0], \text{aux}_1) \\
\text{weightedsumleq}([1,-1], [\text{aux}_1, c_0], \text{aux}_2) \\
\text{weightedsumgeq}([1,-1], [\text{aux}_1, c_0], \text{aux}_2)
\end{align*}
\]

In this example the MINION constraints \( \text{sumleq} \) and \( \text{sumgeq} \) are used to represent the plus operator, and \( \text{weightedsumleq} \) and \( \text{weightedsumgeq} \) together with the given list of signs are for representing the minus operator.

We summarize the conversion of the SSA statements to MINION constraints in Table I using some of the statements from the example given in Fig. 3.

For convenience we assume a function \text{convert} that implements the conversion of programs into MINION constraints as discussed in this section. Hence, \text{convert} takes the number of necessary iterations for each while-statement and the program as input and returns a set of MINION constraints as output. We use this function in the next section, where we discuss an algorithm for computing distinguishing test cases.

### III. COMPUTING DISTINGUISHING TEST CASES

In order to compute distinguishing test cases for two programs \( \Pi_1 \) and \( \Pi_2 \) we have to ensure that the inputs for both programs are the same whereas the computed outputs are different. This idea can be easily represented in MINION. We only have to add the corresponding constraints to the converted programs. Moreover, we have to ensure that the converted programs use different names for the variables. Hence, we have to rename the variables in the constraint representation before putting them together. The following algorithm \text{computeDistinguishingTest} takes care of all of the requirements.

**Algorithm computeDistinguishingTest** (\( \Pi_1, \Pi_2, \#\text{It} \))

**Inputs:** Two programs \( \Pi_1 \) and \( \Pi_2 \) having the same input variables (\( IN \)) and output variables (\( OUT \)), and a maximum number of iterations \( \#\text{It} \).

**Outputs:** A distinguishing test case.

1) Call \text{convert}(\Pi_1, \#\text{It}) and store the result in \( M_1 \).
2) Call \text{convert}(\Pi_2, \#\text{It}) and store the result in \( M_2 \).
3) Rename all variables \( x \) used in constraints \( M_1 \) to \( x_{_{P1}} \).
4) Rename all variables \( x \) used in constraints \( M_2 \) to \( x_{_{P2}} \).
5) Let \( M \) be \( M_1 \cup M_2 \).
6) For all input variables \( x \in IN \) do:
   a) Add the constraint \( \text{eq}(x_{_{P1}}, x_{_{P2}}) \) to \( M \).
7) For all output variables \( x \in OUT \) do:
   a) Add the constraint \( \text{dis}eq(x_{_{P1}}, x_{_{P2}}) \) to \( M \).
8) Return the values of the input variables obtained when calling the MINION constraint solver on \( M \) as result.

The \text{computeDistinguishingTest} algorithm obviously terminates. The given programs and sets are all finite and the conversion terminates. Moreover, the constraint solver also terminates after checking all possible solutions when considering only finite domains. The computational complexity is mainly determined by the constraint solver. The conversion itself is polynomial in the size of the programs. Finding a solution for a finite domain is exponential in the number of used variables.

Note that the whole approach is not necessary restricted to the MINION constraint solver. All discussed steps can be adapted to other constraint solvers. In the following section, we present first empirical results of the approach using MINION.

### IV. EXPERIMENTAL RESULTS

We implemented the discussed approach in Java and applied it on some small Java programs ignoring object-oriented features. For each program we have the original bug-free version, and a set of four mutants obtained by manually injecting different single-faults into the original program. Each program comprises at least one loop structure. We generate the discriminating test cases, i.e., kill the mutants, considering 2, 4 and 7 iterations of each loop statement. We present the obtained results in Table II. All the experiments were performed using an Intel Pentium Dual Core 2 GHz computer with 4 GB RAM. We imposed a limit of two hours in which the mutant should be killed and a distinguishing test case have to be computed. In our experiments no out-of-memory error was encountered. All variables from the tested programs are either of type boolean or of type integer. All integer variables are defined over the finite discrete domain \([-250 \ldots 250]\).

In some cases the inserted fault lead to an infinite execution of the loop structure, e.g., replacing a minus with a plus in a while-structure. Due to the fact that our analysis is based on a static representation of the programs using a fixed number of iterations, the constraint solver is still able to compute an output that satisfies the requirements. But when executing the program and its mutant, the mutant will never stop. Hence, our approach does not require to check program termination for computing test cases.

Another limitation of this approach is that there is no guarantee for computing a solution, i.e., a test case that kills the mutant. In order to identify a faulty statement both the original and its mutant must execute that faulty statement. However there are situations when the faulty statement does not have an influence over the output. In this case a distinguishing test case cannot be determined.

One problem we faced in our experiments was the time needed for computing a solution for some examples. See
for example the results of program GcdATC in Table II where MINION was not able to compute a solution for the versions V3 and V4 within 2 hours. The reason was the huge search space and the fact that no variable ordering was imposed. However, after applying a variable ordering where variables are ordered with respect to their first definition in the program, the situation changed. When using the variable ordering MINION had no problem in killing them in less than a second. Due to this particularity, in our approach we always impose an ordering over the input and output variables. Note that for programs of reduced complexity the variable ordering leads to no gains with respect to time performances.

The obtained results are very promising but further studies have to be performed. In particular, generating test cases for larger programs comprising several thousands lines of code and the ability to handle object-oriented constructs are of interest.

V. RELATED RESEARCH

As already mentioned our distinguishing test cases play an essential role in mutation testing. The distinguishing test cases are those who are able to kill a given mutant. However, in classical mutation testing, the distinguishing test cases are not generated, but a given test suite is assessed with respect to its ability of distinguishing all mutants, i.e. to find all injected faults [20], [19]. It would not be useful to inject faults and then generate a distinguishing test case to find these known faults. However, the idea found application in model-based mutation testing, where distinguishing test cases are generated from mutated models and then executed on an implementation. Very early, Tai and Su [23] proposed algorithms for generating test cases that guarantee the detection of Boolean operator errors in electronic circuits.

Our group used different modeling styles and tools to generate the distinguishing test cases: for OCL pre-postcondition contracts a constraint-solver was used [18], for Spec# contracts we exploited Microsoft’s Z3 SMT-solver [21], for LOTOS protocol specifications first a bisimulation checker [16], then an icco-conformance checker [24] was applied, and finally model checkers served to generate distinguishing test cases for embedded systems [25] and REO coordination models [15]. None of these papers considered programs.

In our theory of mutation testing, we also covered test case generation for programs [17]. However, in this work a different normal form for the constraint solver was proposed. Which one is more efficient is open to future experiments. Most recently, colleagues in Oxford use the CBMC model checker to generate distinguishing test cases from non-deterministic C code generated from Matlab/Simulink [22]. In contrast to our test case generator, CBMC relies on SAT-solving and a different intermediate representation of the programming language.

The closest work to ours is [9], [10]. In these papers the authors described the use of constraint solving for test case generation. They also make use of similar conversion techniques. In contrast to the previous work we are focusing on computing distinguishing test cases to be used for debugging and also for test case generation based on program mutations.

VI. CONCLUSION

In this paper we introduced an approach for generating distinguishing test cases based on constraint satisfaction problems. A distinguishing test case for two programs is a test case that reveals different values for the output variables of the programs when using the same input. The application areas are automated test case generation based on program mutations and fault localization. With respect to mutation testing, the distinguishing test cases serve the following purposes: The tests are defined to kill the set of known mutants, which would not be useful in itself. However, these test cases (1) can be used as a basis for further regression testing, (2) they guarantee a certain coverage in the code (it is well-known that mutation coverage subsumes classical coverage criteria, like e.g. branch coverage), (3) via the mutation testing assumption of the coupling effect, these test cases will detect more subtle faults in the program. With respect to fault localization distinguishing test cases are of interest for reducing the number of fault candidates.

Beside the theory and technique behind our approach we present first empirical results using MINION as the underlying constraint solver. The results are promising and the time required for computing a test case can be neglected. Even in the case of 1,500 constraints and almost 1,000 variables the time required for computing a test case was less than 4 seconds.

ACKNOWLEDGEMENT

The work described in the paper was partially funded by the Austrian Science Fund (FWF) under contract number P20199-N15, and the EU FP7 project MOGENTES ICT-216679, Model-based Generation of Tests for Dependable Embedded Systems.
<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>#I/O</th>
<th>#It</th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
<th>#CO</th>
<th>#VarCO</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultATC</td>
<td>12</td>
<td>2/1</td>
<td>4</td>
<td>Killed(0.07s)</td>
<td>Killed(0.06s)</td>
<td>Killed(0.04s)</td>
<td>Killed(0.03s)</td>
<td>47</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>Killed(0.04s)</td>
<td>Killed(0.08s)</td>
<td>Killed(0.07s)</td>
<td>Killed(0.07s)</td>
<td>87</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>Killed(0.01s)</td>
<td>Killed(0.10s)</td>
<td>Killed(0.11s)</td>
<td>Killed(0.11s)</td>
<td>151</td>
<td>92</td>
</tr>
<tr>
<td>SumATC</td>
<td>13</td>
<td>2/1</td>
<td>2</td>
<td>Killed(0.4s)</td>
<td>Killed(0.03s)</td>
<td>Killed(0.4s)</td>
<td>Killed(0.04s)</td>
<td>49</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>Killed(0.4s)</td>
<td>Killed(0.07s)</td>
<td>Killed(0.49s)</td>
<td>Killed(0.47s)</td>
<td>89</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>Killed(0.67s)</td>
<td>Killed(0.11s)</td>
<td>Killed(0.62s)</td>
<td>Killed(0.09s)</td>
<td>149</td>
<td>94</td>
</tr>
<tr>
<td>MultV2ATC</td>
<td>18</td>
<td>2/1</td>
<td>2</td>
<td>Killed(0.28s)</td>
<td>Killed(0.12s)</td>
<td>Killed(0.21s)</td>
<td>Killed(0.18s)</td>
<td>132</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>Killed(0.34s)</td>
<td>Killed(0.23s)</td>
<td>Killed(0.31s)</td>
<td>Killed(0.31s)</td>
<td>418</td>
<td>238</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>Killed(2.09s)</td>
<td>Killed(2.15s)</td>
<td>Killed(2.15s)</td>
<td>Killed(2.15s)</td>
<td>1144</td>
<td>606</td>
</tr>
<tr>
<td>DivATC</td>
<td>22</td>
<td>2/1</td>
<td>2</td>
<td>Killed(0.06s)</td>
<td>Killed(0.06s)</td>
<td>Killed(0.06s)</td>
<td>Killed(0.06s)</td>
<td>65</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>Killed(0.08s)</td>
<td>Killed(0.08s)</td>
<td>Killed(0.08s)</td>
<td>Killed(0.08s)</td>
<td>105</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>Killed(0.10s)</td>
<td>Killed(0.10s)</td>
<td>Killed(0.09s)</td>
<td>Killed(0.12s)</td>
<td>165</td>
<td>112</td>
</tr>
<tr>
<td>GcdATC</td>
<td>24</td>
<td>2/1</td>
<td>2</td>
<td>Killed(0.07s)</td>
<td>Killed(0.35s)</td>
<td>Killed(0.35s)</td>
<td>Killed(0.35s)</td>
<td>126</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>Killed(0.08s)</td>
<td>Killed(0.08s)</td>
<td>Killed(0.08s)</td>
<td>Killed(0.08s)</td>
<td>206</td>
<td>138</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>Killed(0.10s)</td>
<td>Killed(0.10s)</td>
<td>Killed(0.10s)</td>
<td>Killed(0.10s)</td>
<td>333</td>
<td>220</td>
</tr>
<tr>
<td>RandomATC</td>
<td>52</td>
<td>3/1</td>
<td>2</td>
<td>Killed(0.25s)</td>
<td>Killed(0.25s)</td>
<td>Killed(0.24s)</td>
<td>Killed(0.24s)</td>
<td>305</td>
<td>213</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>Killed(0.8s)</td>
<td>Killed(0.8s)</td>
<td>Killed(0.8s)</td>
<td>Killed(0.8s)</td>
<td>667</td>
<td>433</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>Killed(3.5s)</td>
<td>Killed(3.47s)</td>
<td>Killed(3.6s)</td>
<td>Killed(3.59s)</td>
<td>1513</td>
<td>943</td>
</tr>
</tbody>
</table>

**TABLE II**

For each program NAME, LOC is the number of statements in the original program. #I/O represents the number of inputs and outputs involved in the generated test case. #It represents the number of iterations for the top-down link V1,V2,V3 and V4 designate four different mutants of the program. #CO designates the number of MINION constraints whereas #VarCO designates the number of variables associated to the MINION constraint system. The table indicates the time necessary to identify a suitable test case, able to "kill" the mutant. X stands for not being able to kill the mutant within less than 2 hours.

**REFERENCES**


