Automated Test Generation Survey

Rui Gonçalo

pg18378@alunos.uminho.pt

Techn. Report CROSS-12.12-1
2012, Dezember

CROSS
An Infrastructure for Certification and Re-engineering of Open Source Software
(Project PTDC/EIA-CCO/108995/2008 / FCOMP-01-0124-FEDER-010049)

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR

Centro de Ciências e Tecnologias de Computação (CCTC)
Departamento de Informática da Universidade do Minho
Campus de Gualtar — Braga — Portugal

CROSS-12.12-1
Automated Test Generation Survey by Rui Gonçalo

Abstract

This work presents a bibliographic study of the work published in the area of automated test generation and reports on experiments with the available tools and to produce proofs of concept of how they can be used for generating test cases with high coverage.

1 Introduction

At first, it's important to understand the concept of a software test case. A test is an assignment to the input variables of a program, which allows the tester to verify the behavior of the source code regarding program bugs. When a program is said to be hard tested, means that the program was executed quite often with a collection of values that were assigned to the input variables. It's a fact that the testing phase in software development is a very important process to the final product and entails great cost and time consuming. Despite the testing process is automated in modern development environments, the assignment previously mentioned remains a largely manual process. Safety critical software demands high levels of test coverage, thus an automated test generator able to address this issue becomes mandatory.

Considering the control flow graph of a certain program, there might be branch points guarded by predicates, i. e. in C language, for instance, the `if-then-else` statement creates a branch point in a control flow graph, yielding two branches, when the condition is true and false. A test case results from the assignment of an input that drives execution of the program along a certain path in its control flow graph. Conjoining the predicates of the path we get the condition that the desired test input must satisfy. This process can be effectively performed by model checkers. Model checking is the technique that allows to verify if a given specification holds on a given model. This model represents the behavior of a certain system or program and the specification is formally written, usually, in temporal logic. In test generation, the temporal logic formula used is, commonly, "always not p", where p is the property we want to verify.

There are several model checking algorithms that suits different goals of test generation and programs' characteristics. *Explicit model checking* uses an explicit representation for states and enumerates the set of reachable states by forward exploration until either if finds a violation of the assertion or it reaches a fixed point. To explore the reachable states, two techniques can be applied: *depth-first search* and *breadth-first search*. The former uses less memory and finds counterexamples quickly. The later requires more memory and takes longer, but usually finds the shortest counterexample. Explicit mode checking is useless to test generation using depth-first search, because the counterexample produced are often quite long. And, through breadth-first search is only useful until the number of states to be explored reaches a few million.

Symbolic model checking represents sets of states, functions and relations as propositional formulas, known as binary decision diagrams. The performance of symbolic model checkers is sensitive to the size and complexity of the transition relation and to the size of the total state-space, but is less sensitive to the number of reachable states. For test case generation, the performance is usually good, but when the number of binary decision diagram variables is more than few hundred and when the transition relation is large, symbolic model checkers are unattractive.

Bounded model checkers are specialized to generation of counterexamples. They use a depth bound k and search for a counterexample up to that depth by casting it as a constraint satisfaction problem, applied to SAT solvers that can handle problems with many thousands of variables and constraints. The experiments of [4] confirm the effectiveness of bounded model checkers for test generation.

Rushby, in [7], describes particular cases where other model checking techniques should be adopted. For instance, when the program is not finite state it's recommended to use infinite state bounded model checkers, that use decision procedure for satisfiability modulo theories (SMT) rather than a Boolean SAT solver.

This report is structured as follows: at first, important concepts about test generation and model checking are presented, in the first section. Section 2 states relevant definitions about coverage, particularly, modified condition/decision coverage. A practical example is presented, as well. The

following section describes the use of the CBMC tool to automated test generation. Section 4 presents a different approach concerning test generation through model checking, briefly introducing the SAL platform. Concolic testing is another approach to the subject, and is presented in section 5. Finally, some experimental results are showed, followed by the conclusions.

2 Coverage

This section describes the several notions related to coverage, according to [9], to enhance the relevance of *Modified Condition/Decision Coverage* (MC/DC) stated in DO-178B . By *coverage* we understand a measure to evaluate the level of goals achievement. Test coverage in software measures the amount of source code points a testing process reaches. Within test coverage, two specific measures are distinguish: *requirements coverage* and *structural coverage*. Requirements coverage binds software requirements and test cases, while structural coverage binds the code structure and test cases [5]. The following subsections present the types of structural coverage, where MC/DC is included.

2.1 Statement Coverage

In statement coverage, every statement in the program has been invoked at least once. This criteria is considered the weakest one, because is insensitive to come control structures.

```
1 if ( (x > 1) && (y == 0) )
2   z = z / x;
3
4 if ( (z == 2) || (y > 1) )
5   z = z +1;
```

Listing 1: Control structure

In Listing 1, assigning (x = 2, y = 0, z = 4) as input values, every statement is executed at least once. However, if the `and` operator in the first if-statement gets replaced by an `or` operator, the test case will not notice. That's why statement coverage is insensitive to some control structures and generally considered useless.

2.2 Decision Coverage

In decision coverage, every point of entry and exit in the program has been invoked at least once and every decision in the program has taken all possible outcomes at least once.

```
1 if ( (x == 2) || (y > 1) )
2   x = x +1;
```

Listing 2: Decision with two conditions

In Listing 2, taking (x = 2, y = 1) as input values in the first test case, and (x = 1, y = 1) in the second test case, the result of the decision will be *True* and *False*, ensuring all possible outcomes. However, the effect of the second condition (y > 1) is not tested. The result depends only on the first condition (x == 2).

2.3 Condition Coverage

In condition coverage, every point of entry and exit in the program has been invoked at least once and every condition in a decision in the program has taken all possible outcomes at least once.

Using the example in Listing 2, if the input values are (x = 2, y = 1) for the first test case, and (x = 1, y = 2) for the second, all possible condition outcomes are tested, (*True* or *False*) and (*False* or *True*) However, the result of the decision is *True* in both test cases and all possible decision outcomes are not tested.

2.4 Condition/Decision Coverage

The obvious answer to the weaknesses of condition coverage and decision coverage is to add both in a condition/decision coverage, where every point of entry and exit in the program has been invoked at least once, every decision in the program has taken all possible outcomes at least once and every condition in a decision in the program has taken all possible outcomes at least once.

Once again, using the code in Listing 2, the following input values are assigned: (x = 2, y = 2) and (x = 1, y = 1), deriving in (*True* or *True*) and (*False* or *False*). The decision for the test cases will result in *True* and *False* respectively. However, the decision result is the same as if it was a single decision composed by the first condition (x == 2) or by the second condition (y > 1) or replacing the `or` operator by the `and` operator.

2.5 Modified Condition/Decision Coverage

Finally, ensuring that every point of entry and exit in the program has been invoked at least once, every decision in the program has taken all possible outcomes at least once and every condition in a decision in the program has taken all possible outcomes at least once, is not enough, so also every condition in a decision must be shown to independently affect that decision's outcome. Generally, to get MC/DC, a minimum of n+1 test cases for a decision with n inputs is required.

Considering the same example in Listing 2, the following test cases were needed: (x = 2, y = 1), (x = 1, y = 2) and (x = 1, y = 1), deriving in (*True* or *False*), (*False* or *True*) and (*False* or *False*), resulting in *True*, *True* and *False*, respectively.

2.6 Practical example

Applying the concepts of code coverage described above to a practical case is the best way of understanding. So, consider the source code in Listing 3.

```
1  int func(int x, int y) {
2    int a = 0;
3    if ( x > 3 || y == 1 )
4      a = x + y;
5    else
6      if ( x == y )
7        a = x;
8    a++;
9    return a;
10 }
```

Listing 3: Practical example

Figure 1 presents the correspondent control flow graph. The graph is composed by a set of statements S_{func} = {S_0,S_1,S_2,S_3} and a set of branches B_{func} = {B_0,B_1,B_2,B_3,B_4}. It is possible to trace a path predicate along the execution of the program, depending on the input. For instance, if the input is (x = 0, y = 0), the path will be {S_0,B_4,S_2,S_3} and the path constraint or the path predicate will be $(x \leq 3 \vee y \neq 1) \wedge (x = y)$.

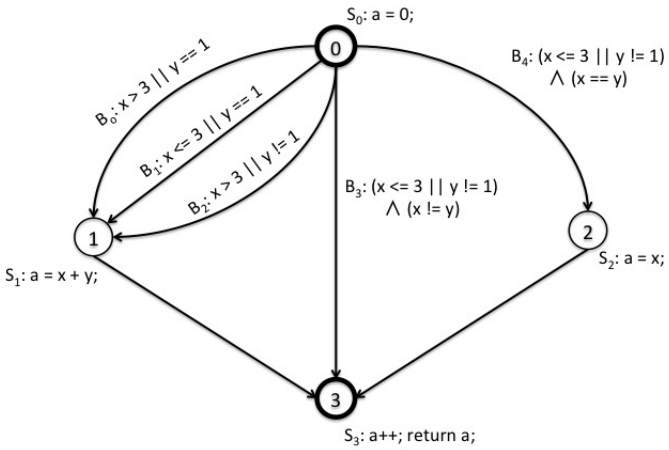

Figure 1: Control flow graph

To fulfil the modified condition/decision coverage, a set of tests $T = \{T_0,...,T_n\}$ must contain all elements of the set of statements and the set of branches. Given

T_0 = (x = 4, y = 0)
T_1 = (x = 4, y = 1)

the following statements and branches are covered:

$S_{T_0} = \{S_0,S_1,S_3\}$ and $B_{T_0} = \{B_2\}$
$S_{T_1} = \{S_0,S_1,S_3\}$ and $B_{T_1} = \{B_0\}$

Adding

T_2 = (x = 3, y = 1)
T_3 = (x = 3, y = 0)
T_4 = (x = 3, y = 3)

the following statements and branches are covered:

$S_{T_2} = \{S_0,S_1,S_3\}$ and $B_{T_2} = \{B_1\}$
$S_{T_3} = \{S_0,S_2,S_3\}$ and $B_{T_3} = \{B_3\}$
$S_{T_4} = \{S_0,S_2,S_3\}$ and $B_{T_4} = \{B_4\}$

Conjoining the sets we get all elements of S_{func} and B_{func}

$S_T = \{S_0,S_1,S_2,S_3\}$
$B_T = \{B_0,B_1,B_2,B_3,B_4\}$

which means that to achieve the MC/DC, we have to execute five test cases, at least.

3 Bounded Model Checking for C programs

The goal of *Bounded Model Checking* is to verify if a given program satisfies a given property, within a given bound. This property is the predicate we want to test. Basically, the program and the property are translated into a Boolean formula in *Conjunctive Normal Form* (CNF) and are passed to a SAT solver. If the solver determines that the formula is unsatisfiable then the property holds (for the given bound), otherwise the property doesn't hold and a counterexample is produced. The set of counterexamples is the set of desired tests.

In order to apply Bounded Model Checking to software verification, the SAT solver must receive the program code in Conjunctive Normal Form. The translation from C code into CNF is accomplished by taking the following steps:

1. Each function call is replaced by its body. Consider the C program in Listing 4;
2. Each loop is unwound, i.e. `while` statements and `for` statements are replaced by `if-then-else` statements, k times (bound). In Listing 5 the bound is *1*, thus the loop is unwound one time, transforming the `while` statement into one `if-then-else` statement, keeping the same decision `(i < max)`;
3. The program and the property are rewritten into an equivalent program in *Single Static Assignment* (SSA) form, where each variable is assigned only once. To achieve this goal, the variables are unfolded into several variables. For instance, given `x = 0` as the first assignment to the variable `x`, and further in the program there is an assignment `x = x + 1`. The SSA form requires the variable `x` to be assigned only once, thus it must be unfolded into new variables x_i, $0 \leq i \leq$ number of times that `x` is assigned. In Listing 6 the variables with replaced values during the program's execution keep the value from the first assignment and new variables are created that will keep the new values from the new assignments.

```
1  int func(int a) {
2    int r = 0, i = 0;
3    while (i < max)
4    {
5      a++;
6      assert( a != 0 );
7      r = max + r / a;
8    }
9    r = r * 2;
10   return r;
11 }
```

Listing 4: C program

```
1  int func(int a) {
2    int r = 0, i = 0;
3    if (i < max)
4    {
5      a++;
6      assert( a != 0 );
7      r = max + r / a;
8    }
9    r = r * 2;
10   return r;
11 }
```

Listing 5: Loop unwounded, $k = 1$

```
1 C :=  r0 = 0 ∧
2       i0 = 0 ∧
3       a1 = a0 + 1 ∧
4       r1 = max0 + r 0 / a1 ∧
5       r2 = i0 < max ? r1 : r0 ∧
6       r3 = r2 * 2 ∧
7 P :=  a1 != 0
```

Listing 6: *SSA* form

3.1 CBMC Approach

Minding the importance of bounded model checking, a robust tool called *CBMC*[1] - *Bounded Model Checker for ANSI-C* has been built. This tool is supported by an assertion generator called

[1] http://www.cprover.org/cbmc/ (Nov 2012)

goto-instrument that produces assertions to determine program locations that potentially contain bugs. These assertions verify the following properties: buffer overflows, pointer safety, division by zero, not-a-number, unitialized local variables and data race. The user is able to perform its own assertions, too. Thus, CBMC is a useful tool to perform test generation, reducing the efforts associated to the testing process.

The methodology stated by [2] to generate test cases from C code, using CBMC, is the following:

1. Ensure that all functions called by the `main` function are completely defined. If the desired entry point is other function, for instance, the function `func`, use the flag `-function func`;
2. To set the input values, use the non-deterministic choice functions CBMC provides, which have the prefix `nondet_`;
3. Insert assertions - `assert(0)` - after each decision block of the control flow graph to ensure the desired coverage;
4. CBMC should run as many times as the number of assertions, with the flag `-D ASSERTION_i` to specify the exact assertion. Each run outputs a new test case;
5. Each `assert(0)` reached will stop the execution of CBMC. So, is necessary to implement conditionals `#ifdef`;
6. Finally, the command line should be:
   `$ CBMC -D ASSERT_i file.c --unwind k --no-unwinding-assertions`

The given bound k should be incremented until the desired coverage is achieved. Taking the program in Listing 3 to generate test cases using CBMC, it should be instrumented as presented in Listing 7.

3.2 Eliminating Redundant Tests

This approach allows CBMC to be used as a practical test generator, but not in an efficient way, because a considerable amount of redundant tests may be created. By redundant tests we consider tests that do not contribute to the coverage goals, i. e. several tests reach the same percentage of code coverage instead of each test covers a different portion of the code. Redundant tests are not easy to detect and imply additional costs during the testing phase. In order to overcome this issue, an effective algorithm was proposed [1]. Considering the control flow graph of a program, the algorithm selects a set of independent paths, i. e. a set of paths in which there is at least a branch in a path that do not occur in any other path of the set.

Basically, the algorithm is composed by three functions - *pathgenerator*, *genapath* and *atgviacbmc*. The `pathgenerator` function takes as input the control flow graph of the program under test and outputs the set of independent paths. Listing 8 presents the algorithm. `C` is the control flow graph, `P` is the set of paths and `A` is the set of branches in `C` not yet covered by any path in `P`. The function `update` returns all the branches of `C` not yet covered by `P`. Thus, at first, `A` is updated with all the branches of `C`. Each iteration of the loop adds a new path to `P` through the function `genapath`, until all branches of `C` are covered by `P`.
The main algorithm to eliminate the redundancy is executed by `genapath` function, in Listing 10. It takes as input one node from the control flow graph and the set of branches not yet covered. If the input node is not the end node, n_e, i. e. there is at least one branch from this node to another, the algorithm considers the set of sequent nodes. Then, the node that has the greatest number of sequent uncovered branches is chosen, and `A` is updated. When the end node is reached, the algorithm stops and the path is constructed in a backtrack way, starting from the last node, n_e, including all the nodes explored by the algorithm.

```
1 int func(int x, int y) {
2   int a = 0;
3
4   if ( x > 3 || y == 1 )
5   {
6     #ifdef ASSERTION_1
7     assert(0)
8     #endif
9     a = x + y;
10  }
11  else
12  {
13    #ifdef ASSERTION_2
14    assert(0)
15    #endif
16    if ( x == y )
17    {
18      #ifdef ASSERTION_3
19      assert(0)
20      #endif
21      a = x;
22    }
23  }
24  #ifdef ASSERTION_4
25  assert(0)
26  #endif
27  a++;
28  return a;
29 }
30
31 int main() {
32
33  int x = nondet_int();
34  int y = nondet_int();
35
36  return func(x,y);
37 }
```

Listing 7: Instrumented code to generate test cases with CBMC

At last, the set of paths is applied as stated in Listing 10. The function takes as input the set of paths, `P`, and the source code of the program, `D`, and outputs the set of generated tests, represented by `T`. For each path, the program is properly instrumented. In Listing 7, an *assert(0)* is added when a branch need to be covered. However, in order to cover the branch, CBMC chooses which path has to follow. So, the program is instrumented with `assume (expression = value)` statements in a certain point, to ensures that CBMC will take the desired path.

4 SAL Approach

The *Symbolic Analysis Laboratory*[2] (SAL) created a language that describes transition systems in terms of initialization and transition commands, and provides a tool suite for abstraction, program analysis, theorem proving and model checking towards the calculation of properties of those systems. At this time, the following tools are available: **sal-wfc**: well-formedness checker, **sal-smc**: symbolic model checker, **sal-deadlock-checker**: deadlock checker, **sal-bmc**: bounded model checker, **sal-inf-bmc**: infinite bounded model checker, **sal-atg**: automated test generator, **sal-sim**: simulator, **sal-wmc**: witness model checker, **sal-path-finder**: random paths generator and **sal-emc**: explicit state model checker.

[2] http://sal.csl.sri.com/ (Nov 2012)

```
1 set_of_paths PATHGENERATOR(C)
2   P = {}
3   A = UPDATE(P,C);
4   while |A| ≠ 0 do
5     P = P ∪ GENAPATH(n,A);
6   return P;
```

Listing 8: Set of paths generator

```
1 path GENAPATH(n,A)
2   if n ={}= n_e then return {n}
3   max = SUCC(n).FIRST();
4   foreach s_i ∈ SUCC(n) do
5     if |A(max)| < |A(s_i)| then
6       max = s_i
7     else if |A(max)| ={}= |A(s_i)| then
8       if E(n,max) ∉ A then
9         max = s_i
10    A = UPDATE(P,C);
11  return {n} ∪ GENAPATH(max,A)
```

Listing 9: Generate paths

Concerning automated test generation, SAL presents a different approach that highlights the efficiency and non-redundancy of test sets. Suppose that a reactive system is composed by four states - `A`, `B`, `C` and `D` - and the transitions $\texttt{t}_{o->d}$ where `o` is the source state and `d` the destination state. The transitions are executed always in the following order - $\texttt{t}_{A->B}$, $\texttt{t}_{B->C}$, $\texttt{t}_{C->D}$ and $\texttt{t}_{D->A}$. If we want to generate a test case that begins in the initial state `A` and exercises the transition $\texttt{t}_{D->A}$, then this test case also exercises the remaining transitions, resulting in much redundancy. SAL proposes an approach that extends existing test cases to reach uncovered goals, rather than start each one again [3].

5 Concolic Testing Approach

Concolic testing combines random testing (**conc**rete execution) and symbolic testing (symb**olic** execution). Symbolic execution was proposed to automate software testing by generating test inputs. Along the program, the primitive variables are not replaced by values (concrete execution) but by symbolic inputs, building a path constraint. Each time a conditional statement is executed, the path constraint is updated. This process goes on until a desired program point is reached, where the constraint is solved using a *contraint solver*. Since the 70s, when symbolic execution was first proposed, that this technique is not widely used. In fact, just recently constraint solvers are powerful enough to be applied in "real world" programs.

The combination of both techniques allows solving path conditions which contain expressions that cannot be handled by constraint solvers, as non-linear constraints or floating-point variables. For instance, let's consider the non-linear condition (x * y $<$ 100). Concrete execution will generate random values to the input variables, in this case, suppose (x = 536, y = 156). The path taking the false branch is executed. To cover the true branch, symbolic execution assigns a symbolic variable y_0, and x is replaced with a value, turning the condition into (536 * y $<$ 100). What was a non-linear condition becomes linear. Therefore, concolic testing assigns a value to y_0 as zero or negative, so that the path can take the true branch. The CUTE tool [8] was built based upon the principles of concolic testing and follows a depth-first strategy.

Consider the code in Listing 11. The function has a state that triggers an error. Instead of testing

```
1 set_of_tests ATGviaCBMC(P,D)
2   T = {};
3   foreach p ∈ P do
4     D' = INSTRUMENT(D,p);
5     T = T ∪ CBMCcall(D');
6     return T;
```

Listing 10: Generate paths

all possible inputs, which is infeasible and most of them result in the same execution path, concolic testing automatically finds the inputs that generate different paths. Suppose the first test assigns random input values (x = 3434, y = 2321). This execution enters in the first conditional, but not in the second. While the test is executed, the condition predicates are recorded in the path constraints. In this case, the path constraint is $\neg(x = y) \wedge \neg(2 * x = x + 10)$. To generate a new input, some predicate is negated. The path constraint becomes $\neg(x = y) \wedge (2 * x = x + 10)$. A possible solution to this constraint is (x = 10, y = 2321), leading to the error in the program. The next input is a solution for (x == y) that might be (x = 10, y = 10). The collection of all path constraints is finished, because all predicates have already been negated [10].

```
1 int aux(int x) {
2   return a * 2;
3 }
4
5 int func(int x, int y) {
6   if (x != y)
7     if(aux(x) == x + 10)
8       error();
9
10  return 0;
11 }
```

Listing 11: Portion of a C program

6 Experimental Results

This section presents the results achieved in several researches [2] [1] [6]. To test the effectiveness of CBMC in automatic test generation, the method explained in subsection 3.1 was experimented on a subset of the modules of the European Train Control System (ETCS) of the European Rail Traffic Management System (ERTMS) source code, an industrial system for the control of the traffic railway, provided by Ansaldo STS[3]. The experiment should achieve the target 100% code coverage, as required by CENELEC standards. It was found that CBMC, for each module, automatically produced a number of test cases close to the double of the number of test cases manually generated by Ansaldo STS. However, the time consumed to manually generate the tests was much larger than the time required by CBMC and both methods reached the desired coverage. CBMC generated 757 tests in less than 4h, while the tests manually generated were 385 in almost 100h.

The algorithm described in subsection 3.2 was applied in the software of Ansaldo STS, but in a higher number of modules. In all cases, this strategy produced a lower number of test cases for the same level of coverage (full coverage). Even comparing to the manual generation of tests of Ansaldo STS, only 2 modules, in 20, exhibited a higher number of test cases. The time spent by this method is, also, very low.

[3] http://www.ansaldo-sts.com/ (Nov 2012)

Concerning the concolic approach for automated test generation, CUTE was run in several C "real world" programs. The tool was applied in its default mode with no specified bound to unfolding loops. The final results showed that CUTE was not able to test a considerable amount of functions, because the test subject did not compile after CUTE's instrumentation and due to some source code issues as void pointers and data structure casts. The coverage reached was only of 50%.

In the SAL platform, the required assertions to produce counterexamples are specified over *trap properties*. These properties are verified through state variables when the execution of the system reaches a certain control point, takes a certain transition or exhibits some other behavior of interest. The method was applied to three different model systems. The results were promising, although the systems used were not critical components of real industry.

7 Conclusion and Related Work

The experimental results concerning the use of CBMC to automatic test generation crushed those from manual generation, mainly, because of the time consumed. Although the naive method to apply CBMC obtained a higher number of test cases than the manual method, the time difference was considerable big. But the algorithm that cut redundant tests presented a lower size of test set in the same time difference. That's why the methodology proposed based on CBMC leads to a dramatic increase in the productivity of the entire Software Development process by substantially reducing the efforts associated to the testing phase.

The concolic approach embodied in the CUTE tool reveals that this method is not suitable for test generation when high levels of coverage need to be reached. The researchers concluded that a lot of challenges remain to be explored concerning automated test generation. Tools need to be more robust to overcome issues as segmentation faults. Also, static and dynamic analysis must be combined in order to tackle problems such as testing uninstrumented code, overcoming limitations of constraint solvers and preventing flat fitness landscapes in dynamic testing.

The SAL approach reveals techniques to be exercised in models of systems, that need to be written in *Scheme Language*. Using the suitably scriptable API of modern model checkers is possible to search at each step for an extension from any known state to any uncovered goal, regarding the desired coverage. The authors refer that the next step is to use their method in cases of industrial scale and to reach MC/DC coverage.

References

[1] Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, Gabriele Palma, and Salvatore Sabina. Improving the automatic test generation process for coverage analysis using cbmc. In *Proceedings of the 16th International RCRA workshop*, RCRA 2009, 2009.

[2] Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, and Salvatore Sabina. Using bounded model checking for coverage analysis of safety-critical software in an industrial setting. *J. Autom. Reason.*, 45(4):397–414, December 2010.

[3] Gregoire Hamon, Leonardo de Moura, and John Rushby. Automated test generation with sal. 2004.

[4] Gregoire Hamon, Leonardo de Moura, and John Rushby. Generating efficient test sets with a model checker. In *Proceedings of the Software Engineering and Formal Methods, Second International Conference*, SEFM '04, pages 261–270, Washington, DC, USA, 2004. IEEE Computer Society.

[5] Kelly J Hayhurst, Dan S Veerhusen, John J Chilenski, and Leanna K Rierson. A practical tutorial on modified condition/decision coverage. *Management*, (May):1–85, 2001.

[6] Kiran Lakhotia, Phil McMinn, and Mark Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In *Proceedings of the 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques*, TAIC-PART '09, pages 95–104, Washington, DC, USA, 2009. IEEE Computer Society.

[7] John Rushby. Verified software: Theories, tools, experiments. chapter Automated Test Generation and Verified Software, pages 161–172. Springer-Verlag, Berlin, Heidelberg, 2008.

[8] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[9] Certification Authorities Software Team. What is a "decision" in application of modified condition/decision coverage (mc/cd) and decision coverage (dc)? Technical report, June 2002.

[10] Ru-Gang Xu. *Symbolic Execution Algorithms for Test Generation*. PhD thesis, 2009.