Automatic generation of high quality test sets via CBMC

Emanuele Di Rosa
DIST - University of Genova
Genova, Italia
emanuele@dist.unige.it
, Enrico Giunchiglia
DIST - University of Genova
Genova, Italia
enrico.giunchiglia@unige.it
, Massimo Narizzano
DIST - University of Genova
Genova, Italia
massimo.narizzano@unige.it
, Gabriele Palma
DIST - University of Genova
Genova, Italia
gabrielepalma82@gmail.com
and Alessandra Puddu
DIST - University of Genova
Genova, Italia
alessandra.puddu@unige.it

Abstract

Software Testing is the most used technique for software verification in industry. In the case of safety critical software, the test set can be required to cover a high percentage (up to 100%) of the software code according to some metrics. Unfortunately, attaining such high percentages is not easy using standard automatic tools for tests generation, and manual generation by domain experts is often necessary, thereby significantly increasing the associated costs. In previous papers, we have shown how it is possible to automatize the test generation process of C programs via the bounded model checker CBMC. In particular, we have shown how it is possible to productively use CBMC for the automatic generation of test sets covering 100% of branches of 5 modules of ERTMS/ETCS, a safety critical industrial software by Ansaldo STS. Unfortunately, the test set we automatically generated, is of lower "quality" if compared to the test set manually generated by domain experts: Both test sets attained the desired 100% branch coverage, but the sizes of the automatically generated test sets are roughly twice the sizes of the corresponding manually generated ones. Indeed, the automatically generated test sets contain redundant tests, i.e. tests that do not contribute to reach the desired 100% branch coverage. These redundant tests are useless from the perspective of the branch coverage, are not easy to detect and then to eliminate a posteriori, and, if maintained, imply additional costs during the verification process.

In this paper we present a new methodology for the automatic generation of "high quality" test sets guaranteeing full branch coverage. Given an initially empty test set T, the basic idea is to extend T with a test covering as many as possible of the branches which are not covered by T. This requires an analysis of the control flow graph of the program in order to first individuate a path p with the desired property, and then the run of a tool (CBMC [6] in our case) able to return either a test causing the execution of p or that such a test does not exist (under the given assumptions). We have experimented the methodology on 31 modules of the Ansaldo STS ERTMS/ETCS software, thus greatly extending the benchmarking set. For 27 of the 31 modules we succeeded in our goal to automatically generate "high quality" test sets attaining full branch coverage: All the feasible branches are executed by at least one test and the sizes of our test sets are significantly smaller than the sizes of the test sets manually generated by domain experts (and thus are also significantly smaller than the test sets automatically generated with our previous methodology). However, for 4 modules, we have been unable to automatically generate test sets attaining full branch coverage: These modules contain complex functions falling out of CBMC capacity. Our analysis on 31 modules greatly extends our previous analysis based on 5 modules, confirming that automatic test generation tools based on CBMC can be productively used in industry for attaining full branch coverage. Further, the methodology presented in this paper leads to a further increase in the productivity by substantially reducing the number of generated tests and thus the costs of the testing phase.

1 Introduction

Testing [3] is the most used technique for software verification: It is easy to use and even if no error is found, it can release a set of tests certifying the (partial) correctness of the system. In the case of safety critical software, the test set can be required to cover a high percentage (up to 100%) of the software code according to some metrics. Unfortunately, attaining such high percentages is not easy using standard automatic tools for tests generation, and manual generation by domain experts is often necessary, thereby significantly increasing the associated costs. In the literature many techniques have been proposed to automatically generate tests aimed to attain full coverage, see, e.g., [5], [17], [16], [8], [12], [15], [4], [2]. In particular, in [2], we have shown how it is possible to automatize the test generation process of C programs via the bounded model checker CBMC. In more details, we have shown how it is possible to productively use CBMC for the automatic generation of test sets covering 100% of branches of 5 modules of ERTMS/ETCS [1], a safety critical industrial software developed by Ansaldo STS. Unfortunately, the test set we automatically generated, is of lower "quality" if compared to the test set manually generated by domain experts: Both test sets attained the desired 100% branch coverage, but the sizes of the automatically generated test sets are roughly twice the sizes of the corresponding manually generated ones. Indeed, the automatically generated test sets contain redundant tests, i.e. tests that do not contribute to reach the desired 100% branch coverage, though they may increase the fault sensitivity of the test set. Still, these redundant tests are useless from the perspective of the branch coverage, are not easy to detect and then to eliminate a posteriori, and, if maintained, imply additional costs during the verification process. Thus, as discussed in [11] when execution is costly or when the automatic generation process is costly, it is not advantageous to first spend resources on generating a large set of tests, only to discard most of them later. Instead, it is much better to just create a small test set with the desired properties in the first place, i.e., to do test suite reduction at creation time.

In this paper we present a new methodology for the automatic generation of "high quality" test sets guaranteeing full branch coverage. Given an initially empty test set T, the basic idea is to extend T with a test covering as many as possible of the branches which are not covered by T. This requires an analysis of the control flow graph of the program in order to first individuate a path p with the desired property, and then the run of a tool (CBMC in our case) able to return either a test causing the execution of p or that such a test does not exist (under the given assumptions). Our approach thus differ from the standard approach (followed, e.g., in [10]) in which, given an already generated test set T and a branch b, a test covering b is generated only if b is not already covered by T: In our approach, we first generate a path p covering as many as possible branches which are not covered by T, and then we try to generate a test covering p.

We have experimented the methodology on 31 modules of the Ansaldo STS ERTMS/ETCS software, thus greatly extending the benchmarking set used in [2]. For 27 of the 31 modules we succeeded in our goal to automatically generate "high quality" test sets attaining full branch coverage: All the feasible branches are executed by at least one test and the sizes of our test sets are significantly smaller than the sizes of the test sets manually generated by domain experts (and thus are also significantly smaller than the test sets automatically generated with our previous methodology): For the 27 modules, we generated a total of 2059 tests instead of the 2661 tests manually generated and the 3768 tests generated with our previous method. However, for 4 modules, we have been unable to automatically generate test sets attaining full branch coverage: These modules contain complex functions (each with more than 300 lines of code and each with loops) and for 11 functions (out of 115) CBMC was not able to terminate with the available resources. Needless to say that these functions cannot be handled also when using our previous methodology.

Our analysis on 31 modules greatly extends our previous analysis based on 5 modules, confirming that automatic test generation tools based on CBMC can be productively used in industry for attaining full branch coverage. Further, the methodology presented in this paper leads to a further increase in the productivity by substantially reducing the number of generated tests and thus the costs of the testing phase.

The paper is structured as follows: we first give some basic notions used throughout the paper. Then we present the algorithm we use for generating the test sets, and we conclude with the experimental analysis.

2 Basic Definitions

A *flow graph* is a directed graph $G = (N,E,n_s,n_e)$ where

- N is the set of *nodes*;
- E is the set of *edges*, i.e., a subset of $N \times N$,
- $n_s \in N$ and $n_e \in N$ are unique *entry* and unique *exit* nodes respectively.

A *control flow graph* is a representation, using graph notation, of all paths that might be traversed by a program during its execution. Each node in the graph represents a *basic block*, i.e. a piece of code without any jump or jump target; jump targets start a block and jumps end a block. Directed edges are used to represent jumps in the control flow. We assume there are two specially designated blocks: the *entry block*, through which control enters into the flow graph, and the *exit block*, through which all the control flows leave. Moreover, for each edge, it is also annotated which branch condition, if any, it represents. We say that a branch predicate *guards* a basic block if the true value of the predicate implies the execution of the block.

Given a control flow graph G, a *base path* p in G is a sequence $p = \{n_1, n_2, ...n_k\}$ of nodes such that $n_1 = n_s$ and for all $i, 1 \leq i \leq k$, $(n_i, n_{i+1}) \in E$. A *path* is a base path in which the last node is n_e. From a notational point of view, we write $E(n_i, n_j) \in p$, to mean that the edge (n_i, n_j) occurs in the path p.

Notice that an assignment to the input variables determines a path along the control flow graph, but the contrary is not always true. We say that a path is *feasible* if there exists an assignment to the program's input X for which the path is traversed during the program execution, otherwise the path is *unfeasible*.

CBMC is a Bounded Model Checker for software verification [6], that takes as input a C program and checks safety properties such as the correctness of pointer constructs, array bounds, and user-provided assertions. Intuitively speaking, given a program C, a property P and a bound k, the verification is done by first (i) unrolling k times the loops in C; then (ii) translating the resulting program without loops and the property into a Boolean formula in Conjunctive Normal Form (CNF); and finally (iii) giving the result to a SAT solver like chaff [13] or MiniSat [9]: If the SAT solver returns false then the property holds, otherwise the property does not hold, given the bound k. In more details, given a program C and a property P, the flow is the following:

1. Each function call in C is replaced by its function body with proper instantiation of the formal parameters;

2. Each loop is unrolled, i.e. the body is duplicated k times —where k is the given bound— each copy of the body being guarded by an if statement that uses the same condition of the loop statement (*goto* statements are unrolled in a similar way).

3. The program and the property are rewritten into an equivalent program in *Single Static Assignment (SSA)* form [7] that is an intermediate representation where each variable is assigned exactly once: Intuitively, each variable x in the program C after the unrolling is split into many versions x_i, one per definition of x, and the use of a variable x in C is substituted by an x_i, which one depending on a condition on the input X of the program and capturing which of the definitions of x gets executed just before the usage under consideration.

4. Given the property P and a program C both in SSA form, each statement in P and C can be interpreted as a constraint. Thus, a run of the program P violating the property corresponds to an assignment satisfying the formula $C \wedge \neg P$. In order to check the satisfiability of $C \wedge \neg P$, each constraint in C and in $\neg P$ is first converted into a bit-vector equation: Each variable is represented by a bit-vector of fixed size, operations are converted into bit-vector operations, and the result is a Boolean formula whose variables are the bits of the vectors representing the variables in C and P (indeed, additional defined variables may be introduced in the various intermediate steps).

5. Finally, the Boolean formula is converted into Conjunctive Normal Form (CNF), using well known clause conversion methods (see, e.g., [14]), and a SAT solver is invoked:

   (a) If the SAT solver returns that the formula is satisfiable, then the property does not hold, and an assignment to the Boolean variables making the formula true is returned: Starting from such assignment, it is possible to construct an error trace showing where the property does not hold in the program.

   (b) Otherwise, the resulting CNF is false and the property holds for the given bound k: However, we cannot conclude that the property holds for the program, since choosing another $k' > k$ may lead to a property violation.

In order to use CBMC as a test generator a possible approach is to instrument the code in input as in [2] by introducing an *assert(0)* in the basic block we want to cover. The instrumented code is then given as input to CBMC which (unless the block is unfeasible or the program is outside of CBMC capacity) returns an assignment to the input variables (a test) violating the property (*assert(0)*). In our previous work the assertions were inserted (and then activated one by one) after each branch in order to have full branch coverage: In this way we can verify whether it is possible to cover each single branch, but in isolation, without taking into account that a test may cover other branches than that for which it was generated. As result, such methodology may generate more tests than necessary, as shown by the following example:

if a == 10 **then** B_1

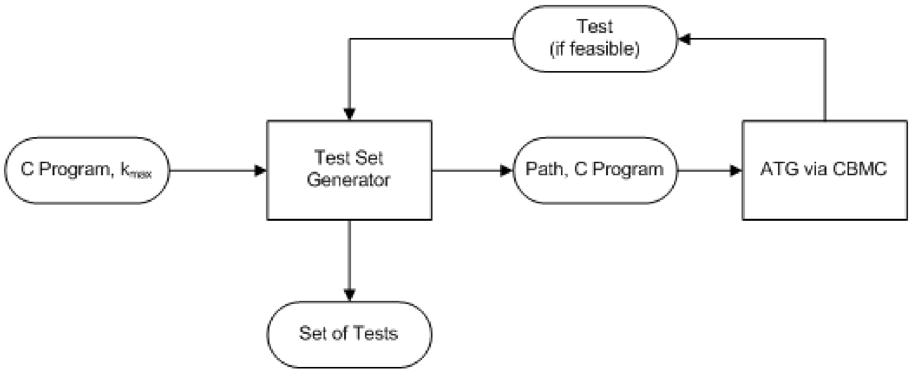

Figure 1: The testing process.

else B_2
....
if a $\neq$ 5 **then** B_3
else B_4

given the methodology in [2], we will introduce an *assert(0)* at the end of each block. For each *assert(0)*, we generate a test, and the resulting test set T $= \{t_1, t_2, t_3, t_4\}$ is guaranteed to cover 100% of the branches by construction. A possible value for the tests in T is:

t_1: (a = 10), t_2: (a = 6), t_3: (a = 5), t_4: (a = 4) .

However it may be the case that eliminating one or two tests from the set T, we still obtain 100% branch coverage: In our example, eliminating t_2 and t_4 leads to the test set $T' = \{t_1, t_3\}$ which still covers 100% of the branches. Indeed, it is possible to first generate a test set T and then try to reduce it by eliminating redundant tests. However, as argued in [11], when execution is costly or when the automatic generation process is costly, it is much better to just create a small test set with 100% branch coverage properties in the first place.

3 Automatic Test Generation via CBMC

Figure 1 shows the testing process, which, at an high level of abstraction, consists of a simple closed loop. Intuitively, given a program C and the maximal bound k_{max} representing the maximal unwinding for CBMC, the function TestSetGenerator is invoked. TestSetGenerator uses a function GeneratePath to select a "best" path p of C which is then passed to the function ATGviaCBMC in order to check the feasibility of p:

1. If ATGviaCBMC returns a test t, t is added to the test set to be returned by TestSetGenerator, a new "best" path is selected and the procedure is iterated by calling ATGviaCBMC.
2. If ATGviaCBMC returns that the path is unfeasible, TestSetGenerator stores this information in order to avoid the further generation of the same path, generates a new "best" path and the procedure is iterated calling ATGviaCBMC.

The loop will ends —returning the set of generated tests— when full coverage is reached or when the system is unable to reach full coverage given the maximal allowed unwinding k_{max}. In

```
path GeneratePath($A_u$, $b_p$, $F$, $G$)
1   $n$ = tail($b_p$)
2   if $n \equiv n_e$ then return $b_p$
3   $candidates$ = SUCC(n) - $\{n_f : E(n, n_f) \in$ F($b_p$)$\}$
4   if $|candidates| == 0$ return [ ]
5   $max_n = candidates$.FIRST();
6   foreach $n_i \in candidates$ do
7     if $(h_1(E(n, max_n), A_u) < h_1(E(n, n_i), A_u))$ then
8       $max_n = n_i$
9     if $(h_1(E(n, max_n), A_u) == h_1(E(n, n_i), A_u))$ then
10      if $((h_2(E(n, max_n)), A_u) < h_2(E(n, n_i), A_u))$ then
11        $max_n = n_i$
12  if $(h_1(E(n, max_n)) == 0)$ then
13    if $(\forall Edge\ e \in b_p, e \notin A_u)$ then
14      return [ ]
15  $A_u = A_u$ - $\{E(n, max_n)\}$
16  return GeneratePath($A_u$, [$b_p$, $max_n$], $F$, $G$)]
```

Figure 2: Function GeneratePath

the following, we detail the description of the functions GeneratePath, TestSetGenerator and ATGviaCBMC.

3.1 GeneratePath

GeneratePath is shown in Figure 2. GeneratePath takes as input:

1. the set of branches which are still to be covered (A_u),
2. a starting base path (b_p),
3. a data structure (F) which given a feasible base path b'_p stores a set S of branches such that if we extend the base path b'_p with a branch in S, we obtain a new base path which cannot be further extended in order to cover at least one branch in A_u. Sometimes it is also notationally useful to view F as a function, which, given a base path, returns the set of *forbidden branches*. For this reason, we may write that $F(b'_p)$ is equal to a set of branches, to mean that F stores the corresponding information.
4. the program's control flow graph (G),

GeneratePath returns either a path p which extends b_p and which covers at least an uncovered branch in A_u, or the empty path if no such path p exists or all candidate paths —extending b_p and covering at least a branch in A_u— are already known to be unfeasible.

The algorithm behaves as follows (n is initialized to the last node in the base path b_p):

1. If n is the end node of G, the base path is complete and thus it is returned on line 2.
2. If not, we consider a set of candidates node, obtained by considering all the possible successors of the current node n minus all such successors reached by using a forbidden branch (line 3).

3. If no candidates are available, no usable path exists from b_p and an empty path is returned (line 4).

4. If more than one candidate is available, the path is extended following the edge e selected on the basis of h_1 and h_2 defined by

$$h_1(e, A_u) = |reach_c(e) \cap A_u|$$
$$h_2(e, A_u) = |reach_a(e) \cap A_u|$$

   where $reach_c(e)$ is the set of edges which are reachable from e, while $reach_a(e)$ is the set of edges which are reachable from e without using a backloop (i.e. an edge pointing to an ancestor of e in the control flow graph). Both $reach_c(e)$ and $reach_a(e)$ are computed only once at the beginning, before any unwinding. Intuitively,

   (a) $h_1(e, A_u)$ gives an indication of the possibility to cover the branches in A_u when taking into account the whole structure of the control flow graph.

   (b) $h_2(e, A_u)$ is used to break ties for edges having a same h_1: Such ties frequently occurs, e.g., when we are considering edges in a loop.

   The functions h_1 and h_2 are used at lines 6-11 to select the "best" candidate according to the above specified criteria.

5. max_n is the candidate node selected at lines 5-11: If neither we have any chance to reach an uncovered edge when making future choices (line 12) nor the base path we are trying to extend covers a branch in A_u(line 13), the path we are going to generate is useless and thus we return the empty path (line 14).

6. Finally, (lines 15, 16), max_n is chosen, A_u is updated possibly removing the chosen edge, and GeneratePath is recursively called using the new base path and the updated A_u.

3.2 TestSetGenerator

TestSetGenerator is presented in Figure 3: Given a program C and the maximal bound k_{max}, TestSetGenerator returns a set of tests which (assuming CBMC does not fail) is guaranteed to cover all the feasible branches when loops are unrolled up to k_{max}.

The algorithm behaves as follows:

1. First, in lines (1)-(5), G_C is initialized to the control flow graph of C; A_u to the set of branches in G_C; the bound parameter k is initially set to 4 which is then used to construct the control flow graph of the unwound program; T stores the test set to return and is initially empty; the base path b_p is initially the starting node and F is the data structure used to compute the forbidden branches.

2. The main loop starts at line 6 and ends when full coverage is reached or current k is greater than k_{max}. At each iteration, a new path is generated by calling GeneratePath (line 7) with the current b_p, A_u and F.

3. If such a path p exists, the feasibility of p is checked by calling ATGviaCBMC (line 9):

```
set of tests TestSetGenerator($C, k_{max}$)
1  $G_c$ = ControlFlowGraph($C$)
2  $A_u = \{e : e \in G_c.E\}$
3  $k = 4$;
4  $G$ = UnwindGraph($G_c, k$)
5  $T = \{\}$; $b_p = \{G.n_s\}$; $F = \{\}$
6  while $|A_u| > 0$ and $k < k_{max}$ do
7    $p$ = GeneratePath($A_u, b_p, F, G$)
8    if $|p| > 0$ then
9      $t$ = ATGviaCBMC($C, p$)
10     if $t == t_0$ then
11       $\langle b_p, F\rangle$ = AnalyzePath($b_p, p, F, G$)
12     else
13       T = T $\bigcup$ {t}
14       $A_u = A_u$ - $\{edge : edge \in p\}$
15       $b_p = \{G.n_s\}$
16   else
17     if $b_p \neq \{G.n_s\}$ then
18       $f_n$ = tail($b_p$)
19       $b_p = b_p \setminus$ tail($b_p$)
20       SetForbidden($F$, $b_p$, E(tail($b_p$), $f_n$))
21     else
22       $G$ = UnwindGraph($G_c, k * 2$)
23       k=k*2; F ={}; $b_p = \{G.n_s\}$
24 return T
```

Figure 3: Function TestSetGenerator

(a) If such a test t exists, (*i*) t is added to the set T of tests to return (line 13); (*ii*) A_u is updated by removing edges covered by t (line 14), and (*iii*) the base path it reinitialized to the starting node (i.e. $b_p = G.n_s$) in order to possibly generate a test covering a possibly completely different path (line 15).

(b) If no such a test t exists (line 11) the path is analyzed by calling AnalyzePath which will return the longer feasible base path in p (which will be our new base path b_p) while the immediately subsequent edge is added to the forbidden branches of our new b_p (i.e., to $F(b_p)$): The next candidate path will be generated starting from b_p and avoiding the known unfeasible paths.

4. If the path generated in line 6 is empty, e.g., because it does not exist a path starting from b_p and covering at least a branch in A_u, then

   (a) assuming the base path is not the initial node of the control flow graph (line 17), (*i*) the last node f_n is removed from the base path (line 19); and (*ii*) the edge leading to the node f_n is added to the forbidden edges of the base path (SetForbidden(F, b_p, E(tail(b_p), f_n)).

   (b) otherwise, we cannot extend the current test set in order to cover branches in A_u and the only option is to increase the bound k: In our procedure, we double its value (lines 22, 23), the loop is iterated and the search restarts again.

```
int TEST(int[ ] a, int size)
s0         int b = 0; int c = 0;
b1, b2, s1 for ( ; (c < size); c++)
b3, b4        if (a[c] < 0)
s2               b++;
s3         return b;
```

```
int TEST(int[ ] a, int size)
s0    int b = 0; int c = 0;
b1    ASSUME(c < size);
b3    ASSUME(a[c] < 0);
s3      b++;
s4      c++;
b1    ASSUME(c < size);
b3    ASSUME(!(a[c] < 0));
s8      c++;
b1    ASSUME(!(c < size));
      ASSERT(0);
s13   return b;
```

Figure 4: A function used as example (left), and its instrumentation (right)

3.3 ATGviaCBMC

ATGviaCBMC is a simple function that takes as input a path and the program and returns (if it exists) a test covering the path. This is obtained by instrumenting the program C in a way and then calling CBMC. In [2], the instrumentation consisted in adding an *assert(0)* in the branch to be covered. However, in this way, CBMC can arbitrarily choose which path to follow in order to generate a test covering the assertion. We can still use CBMC but we need something forcing CBMC to reach an assertion through a specified path. CBMC has a construct, namely

assume (expression = value)

that inserted at a given point in the code enforces the expression to assume exactly that value at that point. Using the *assume* construct we can enforce a path by forcing the branching conditions to assume the desired value, and inserting an *assert(0)* at the end of the program forces CBMC to output a test covering the branch.

3.4 Example

An example of the behavior of our procedures is presented below. Let us assume that the function under test is the one in Figure 4 which counts the negative numbers in an array. The function is quite simple and does not contain unfeasible paths. The algorithm presented above takes as input the program and creates its unwound control flow graphs as shown in Figure 4, left. The set of tests and F are initially empty and A_u is passed with all the edges of the control flow graph. To keep things simple we only consider edges which actually represents decision branches (considering the other branches is not necessary). Thus, $A_u = \{b_1, b_2, b_3, b_4\}$. Additionally, before unwinding occurs, reachability sets $reach_c$ and $reach_a$ are computed as follows: $reach_c(b_1) = reach_c(b_3) = reach_c(b_4) = \{b_1, b_2, b_3, b_4\}$, $reach_c(b_2) = reach_a(b_2) = \{b_2\}$, $reach_a(b_1) = \{b_1, b_3, b_4\}$, $reach_a(b_3) = \{b_3\}$, $reach_a(b_4) = \{b_4\}$. Notice that the function GENERATEPATH works on the unwound control flow graph, which is represented in Figure 5 right, assuming (again for sake of simplicity) $k = 3$ (instead of 4 as in our procedure).

TESTSETGENERATOR after the initialization of the variables, enters the main loop and then calls GENERATEPATH(A_u, [s_0] ,F, G). Initially, there is only one branch exiting from the base path [s_0] and then the recursive call to GENERATEPATH(A_u, [s_0, s_1] ,F, G) is executed. At this point since s_1 is not the end node its successor nodes (s_2 and s_{13}) are explored.

```
**int** TEST($int$[ ] $a$, $int$ $size$)
$s_0$              **int** $b = 0$; **int** $c = 0$;
$s_1,b_1,b_2$      **if** $(c < size)$
$s_2,b_3,b_4$        **if** $(a[c] < 0)$
$s_3$                  $b{+}{+}$;
$s_4$                $c{+}{+}$;
$s_5,b_1,b_2$        **if** $(c < size)$
$s_6,b_3,b_4$          **if** $(a[c] < 0)$
$s_7$                    $b{+}{+}$;
$s_8$                  $c{+}{+}$;
$s_9,b_1,b_2$          **if** $(c < size)$
$s_{10},b_3,b_4$         **if** $(a[c] < 0)$
$s_{11}$                   $b{+}{+}$;
$s_{12}$                 $c{+}{+}$;
$s_{13}$           **return** $b$;
```

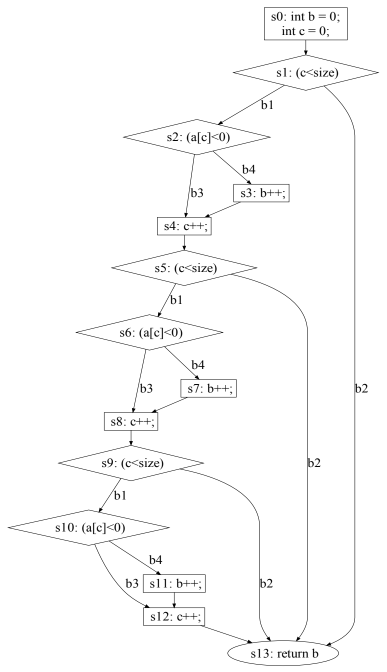

Figure 5: An example of an unwound function and its flow graph.

$max_n = s_2$, since $h_1(b_2) = 1$ and $h_1(b_1) = 4$ (indeed $reach_c(b_1) \cap A_u = \{b_1, b_2, b_3, b_4\}$). The current coverage status is then updated by deleting b_1, which is the incoming edge used to reach s_2, from A_u. A new recursive call, GENERATEPATH(A_u, $[s_0, s_1, s_2]$ F, G) is executed. Then the algorithm has to choose between s_3 and s_4, having a tie on both heuristics: $h_1(b_3) = h_1(b_4) = 3 = |\{b_2, b_3, b_4\}|$, $h_2(b_3) = h_2(b_4) = 1$, the algorithm selects the first node s_3. From node s_3 to node s_5 there is a single path to be followed. On node s_5, $max_n = s_6$ with $h_1(b_1) = 2$ against $h_1(b_2) = 1$.

Between s_7 and s_8, s_8 is chosen thanks to h_2: $h_1(b_3) = h_1(b_4) = 2, h_2(b_3) = 1 > h_2(b_4) = 0$. Finally the algorithm goes from s_8 to s_9 and then to s_{13}, covering b_2. As soon as the algorithm reaches the s_{13} node it stops returning the built path, i.e. $p_1 = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_8, s_9, s_{13}\}$. Figure 4, right, shows the instrumented code generated by ATGVIACBMC to test the generated path. Notice that:

- the instrumented code is already unwound
- unnecessary code is removed in order to help CBMC to find a solution. Since we are exploring a path only the code explored by the path is necessary.

CBMC, taking as input the modified code, generates a test t, e.g., corresponding to the following inputs: $size = 2$, $a[0] = -1$ and $a[1] = 0$. The path covers all the branches, so no more calls are necessary. The test is added to the test set and A_u is updated in TESTSETGENERATOR which therefore exits the loop and returns the set of tests $T = \{t\}$.

		Man.	Naive	Automatic						
Name	#F	#T	#T	#T	#UF	#TO	%C	%C$_f$	$\frac{M-A}{A}$	$\frac{N-A}{A}$
Mod.01	8	27	44	18	0	0	100%	100%	50%	144%
Mod.02	16	144	182	108	0	0	100%	100%	33%	69%
Mod.03	1	17	12	6	0	0	100%	100%	183%	100%
Mod.04	1	43	83	41	0	0	100%	100%	5%	102%
Mod.05	2	18	76	7	0	0	100%	100%	157%	986%
Mod.06	8	39	123	27	0	0	100%	100%	44%	356%
Mod.07	26	127	154	106	0	0	100%	100%	20%	45%
Mod.08	24	200	280	148	0	0	100%	100%	35%	89%
Mod.09	12	57	149	43	0	0	100%	100%	33%	247%
Mod.10	9	35	98	38	0	0	100%	100%	-8%	158%
Mod.11	22	312	340	263	0	0	100%	100%	19%	29%
Mod.12	12	34	35	32	0	0	100%	100%	6%	9%
Mod.13	11	181	210	156	0	0	100%	100%	16%	35%
Mod.14	11	157	196	132	0	0	100%	100%	19%	48%
Mod.15	29	322	335	238	0	0	100%	100%	35%	41%
Mod.16	14	69	96	29	0	0	100%	100%	138%	231%
Mod.17	7	36	73	28	0	0	100%	100%	29%	161%
Mod.18	26	101	170	82	0	0	100%	100%	23%	107%
Mod.19	8	62	89	38	0	0	100%	100%	63%	134%
Mod.20	25	119	161	110	0	0	100%	100%	8%	46%
Mod.21	11	42	99	38	1	0	99%	100%	11%	161%
Mod.22	9	43	72	35	2	0	97%	100%	23%	106%
Mod.23	13	80	99	46	2	0	98%	100%	74%	115%
Mod.24	19	77	143	59	1	0	99%	100%	31%	142%
Mod.25	18	143	184	106	1	0	99%	100%	35%	74%
Mod.26	3	27	35	9	1	0	97%	100%	200%	289%
Mod.27	23	149	230	116	1	0	99%	100%	28%	98%
Mod.28	21	45	92	45	2	3	63%	62%	0%	104%
Mod.29	7	15	54	14	0	2	60%	60%	7%	286%
Mod.30	9	29	65	23	2	2	66%	64%	26%	183%
Mod.31	10	107	124	33	0	4	53%	53%	224%	276%
Total	415	2857	4103	2174	13	11	95%	94%	31%	89%

Table 1: Experimental comparison on 31 modules of Ansaldo STS ERTMS/ETCS software

4 Experimental Analysis

Nowadays trains are equipped with up to six different navigational systems which are extremely costly and take space on-board. A train crossing from one European country to another must switch the operating standards as it crosses the border. The European Rail Traffic Management System [1] is an EU "major European industrial project" to enhance cross-border interoperability and signaling procurement by creating a single Europe-wide standard for railway signaling. ERTMS has two basic components:

- the European Train Control System (ETCS), which transmits speed information to the train driver and it monitors constantly the driver's compliance to the speed limits;

- the radio system based on the standard GSM, used to exchange voice and data information between the track and the train.

Ansaldo STS, as part of the mentioned European Project, produces the European Vital Computer (EVC) software, a fail-safe system which supervises and controls the speed profiles using the information received from the in-track balises transmitted to the train. Following the CENELEC standards Ansaldo STS needs to provide a certificate of the integrity level required, i.e. it has to provide a set of tests covering the 100% of the branches. In order to simplify the readability, the Ansaldo STS implementation of the EVC is developed into different modules of almost fixed size. In our experimental analysis we took a subset of the interconnected modules of the EVC and we applied the automatic test generation strategy described in the previous section. We got more than 130 different modules, containing more than 100.000 lines of code distributed in more than 1700 functions. For benchmarking our procedure, we considered 31 different modules with the following features:

- about 30.000 lines of code, written in ANSI C
- presence of nested structures (struct)
- presence of arrays and pointers
- presence of function pointers
- presence of almost all the constructs of the C language
- presence of nested loops and switches
- absence of the goto construct
- absence of recursive calls.

All the experiments have been run on a Linux box equipped with a Pentium IV 3.2GHz processor and 1GB of RAM. The time limit for CBMC has been set to 1200 s.

Table 1 shows the results, where the columns, from left to right, have the following meaning:

1. Name: The name of the module (for copyright reasons they are coded),
2. #F : the number of functions contained in the module,
3. Man. #T: the number of tests manually generated by domain experts, provided by Ansaldo STS,
4. Naive #T: the number of tests generated while trying to reach the 100% of branch coverage using the naive method presented in [2],
5. Automatic: the data obtained while trying to reach the 100% of branch coverage using the methodology presented in this paper. The results that we present are:
   (a) #T: the number of tests generated,
   (b) #UF: The number of functions for which we did not obtain full branch coverage,
   (c) #TO: The number of CBMC timeouts,
   (d) %C: the branch coverage, i.e., the fraction between the number of branches which are executed by at least one test and the total number of branches in the module,

(e) %C$_f$: the feasible branch coverage, i.e., the fraction between the number of branches which are executed by at least one test and the total number of *feasible* branches in the module,

(f) $\frac{M-A}{A}$ (M is the number in column "Man. #T", A is the number in column "Automatic #T"): The fraction between $(M-A)$ and A in percentage, i.e., how bigger is in percentage the test set manually generated by Ansaldo STS, when compared to the tests set we generate,

(g) $\frac{N-A}{A}$ (N is the number in column "Naive #T", A is the number in column "Automatic #T"): The fraction between $(N-A)$ and A in percentage, i.e., how bigger is in percentage the test set generated according to [2], when compared to the tests set we generate.

Notice that in the table we do not report the branch coverage percentage "#C" and the feasible branch coverage percentage "#C$_f$" for the test set generated either manually or as in [2]: Indeed, the manually generated test set has always #C$_f$= 100%; while there is no difference between the values of #C and #C$_f$ when the test set is generated as in [2] or as in this paper (in other words, our procedure fails exactly on the branches on which [2] fails). Further, we do not show the time needed to generate the tests: As already described in [2], the total time needed to automatically generate the tests is very low and it is not comparable with the time spent by Ansaldo STS to manually generate them (estimated in 15 minutes for each single test).

In the following, for brevity, we call "manual" (resp. "naive") the manual generation by Ansaldo STS (resp. the automatic generation as in [2]).

Considering the results in the table, the first observation is that our algorithm on the first 27 modules always reaches 100% of the feasible branch coverage. Notice that for Mod.01 to Mod.20, #C = #C$_f$ = 100%, while Mod.21 to Mod.27 have some branches which are unfeasible: These branches have been discovered to be unfeasible by manual inspection, and have been introduced in the program according to a defensive style of programming. Further, the number of tests we generated is always (but for Mod.10 in the case of the manual method) smaller than then number of tests manually or naively generated: Looking at the totals, the manual (resp. naive) method produces 31% (resp. 89%) more tests[1]: Thus, our approach succeed in generating a significantly smaller test set than the naive approach, smaller also than the manual approach.

Looking at the results for Mod.27 to Mod.31, for them we have been unable to obtain full coverage because CBMC times out in a few cases when dealing with some particularly complex functions, consisting of some hundreds lines of code and loops which, when unrolled, lead to code consisting of several thousands lines of code. The functions in these modules on which CBMC did not time out, have been fully covered, with less tests than the manual and naive generation.

5 Conclusions

In this paper we have presented a methodology for the automatic generation of test sets for full branch coverage. We have experimented our methodology on 31 modules of the ERTMS/ETCS source code, an industrial system for the control of the traffic railway, provided by Ansaldo STS. With our methodology we have been able to generate a test set (*i*) which fully covers the branches of 27 of the 31 modules we considered, and (*ii*) with size significantly smaller than

[1]In the case of the manual method it can be objected that these figures are not fair because for the manual case it includes a set T of tests covering branches not covered by the naive and our method. However, if we restrict to the first 27 modules or if we do not count the tests in T for the manual case, we obtain an increase of roughly the 30%.

the size of the test set generated according to [2] or manually generated by domain experts. The 4 modules for which we have been unable to automatically obtain a test set covering all the branches contains complex functions which fall outside of CBMC capacity.

Our analysis, significantly extending the one in [2], gives further evidence that model checking can be productively used for automatic test generation in industry, in many cases replacing the very costly manual generation of tests by domain experts. The methodology we presented, is a significant improvement over [2] because it leads to a significant reduction in the size of the automatically generated test set, obtaining the same branch coverage. As we already said in the introduction, having a smaller test set is important when testing is costly.

References

[1] ERTMS: The official website., 2008. `http://www.ertms.com`.

[2] Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, and Salvatore Sabina. Automatic test generation for coverage analysis of ERTMS software. In *ICST*, pages 303–306, 2009.

[3] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[4] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[5] Hana Chockler, Orna Kupferman, Robert P. Kurshan, and Moshe Y. Vardi. A practical approach to coverage in model checking. In *CAV*, pages 66–78, 2001.

[6] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *Proc. TACAS*, pages 168–176, 2004.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[8] Jonathan de Halleux and Nikolai Tillmann. Parameterized unit testing with pex. In *Proc. TAP*, pages 171–181, 2008.

[9] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pages 502–518, 2003.

[10] Gordon Fraser and Franz Wotawa. Using LTL rewriting to improve the performance of model-checker based test-case generation. In *Proc. A-MOST*, pages 64–74, 2007.

[11] Gordon Fraser, Franz Wotawa, and Paul Ammann. Issues in using model checkers for test case generation. *Journal of Systems and Software*, 82(9):1403–1418, 2009.

[12] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[13] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[14] D.A. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation*, 2:293–304, 1986.

[15] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[16] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In *Proc. TAP*, pages 134–153, 2008.

[17] Vivekananda Murthy Vedula. *Hdl slicing for verification and test.* PhD thesis, 2003. Supervisor-Abraham, Jacob A.