Improving the Automatic Test Generation process for Coverage Analysis using CBMC

Damiano Angeletti[1], Enrico Giunchiglia[2], Massimo Narizzano[2], Gabriele Palma[2], Alessandra Puddu[2], and Salvatore Sabina[1]

[1] Ansaldo STS
Via Paolo Mantovani, 3 16151 Genova, Italy
name.surname@ansaldo-sts.com

[2] DIST
University of Genoa
Via all'Opera Pia 13 16145 Genova, Italy
name.surname@unige.it

Abstract

Software Testing via Coverage Analysis is the most used technique for software verification in industry, but, since manual generation is involved, remains one of the most expensive process of the software development. Many tools have been proposed for the automatic test generation: on one hand they reduce the cost of the testing generation process, on the other hand the quality of the test set so generated is lower than the quality of the test set manually generated by domain experts. In fact, most of the time, the test set automatically generated contains redundant tests, i.e. test that does not contribute to reach the property of 100% of coverage: if they are eliminated by the set, the property still holds. Indeed, these redundant tests are useless from the perspective of the 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 methodology for the automatic generation of test set for Coverage Analysis, containing only non redundant test. We experimented it on a subset of modules of the ERTMS/ETCS source code, an industrial system for the control of the traffic railway, provided by Ansaldo STS, where we were able to verify completely 31 different modules of the ERTMS: On 20 modules we were able to generate a set of minimal test covering the 100% of the code for each function in each module; on 7 of them we generate a set of tests that does not cover the 100% of the branches due to unreachable code; the last 4 modules were not completely verified due to CBMC exponential blow-up explosion. The use of our methodology for test generation led to a dramatic increase in the productivity of the entire Software Development process by substantially reducing the number of tests generated and thus the costs of the testing phase.

Proceedings of the 16th International RCRA workshop (RCRA 2009):
Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion
Reggio Emilia, Italy, 12 December 2009

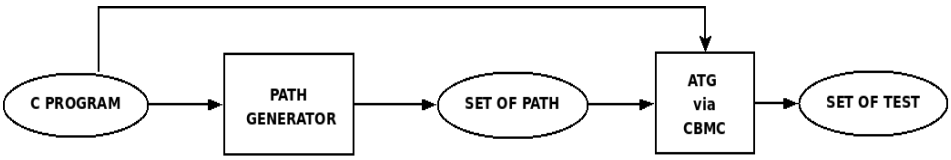

Figure 1: testing process

1 Introduction

Testing [8] 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 compiled system. The most used technique for test generation is random testing since it is automatic and simple to apply. However, it does not ensure an extensive test of the code: since it merely relies on probability it has quite low chance of finding semantically small faults. Thus, in order to increase the confidence of the correctness of the compiled system, it is often required that the provided set of tests covers 100% of the code. This requirement, however, substantially increases the costs associated to the testing phase, since it may involve the manual generation of tests. In the literature many techniques have been proposed to automatically generate test for coverage analysis [9], [10], [7]: on one hand they reduce the cost of the testing generation process, on the other hand most of the times the quality of the test set so generated is lower than the quality of the test set manually generated by domain experts. For instance, in a previous paper [7] we have shown how it is possible to automatically generate test sets covering 100% of the code using a bounded model checker in an industrial setting: Such test set met the desired coverage properties previously reached only by manual generation, but on the other hand it contained many redundant tests, i.e., tests not contributing to covering not otherwise covered portions of the code. Indeed, these redundant tests are useless from the perspective of the coverage, are not easy to detect and then to eliminate a posteriori, and, if maintained, imply additional costs during the verification process. Determine which test set to use is still a major problem in program testing. One technique that is in widespread use is to take the control flow graph from each of the program function and calculate a set of independent paths, i.e. a set of paths through the functions that are linearly independent. From this set, that has to cover the 100% of the branches in the control flow graph, it is possible to construct a test exercising that path, unless the path is unfeasible. In this paper we present a methodology for the automatic generation of a set of non redundant test, through the construction of a set of independent paths covering the 100% of the branches in the control flow graph of the function under test. In other words, given a program to test, for each function we firstly construct the

control flow graph from which we deduce an independent set of paths. Then for each path we generate a test that, when the program (function) is executed, it explores exactly that path. As a test generator we use CBMC [1] a Bounded Model Checker for low-level ANSI-C programs. CBMC checks safety properties such as the correctness of pointer constructs, array bounds, and user-provided assertions. CBMC, as showed in [7], can also be used as a test generator, if it can take as input an instrumented code. However, the methodology proposed in [7] does not generate a set of tests without redundancy: usually the path explored to generate a test is chosen by CBMC in a heuristic way, thus the set of paths associated to the set of tests generated by CBMC is not guarantee to be non redundant. In this paper we show also how to instrument a function in order to force CBMC to generate a test following a given path. In this way we can generate a set of tests that satisfies the 100% of the branch coverage. We experimented our methodology on a subset of modules of the ERTMS/ETCS [5] source code, an industrial system for the control of the traffic railway, provided by Ansaldo STS. ERTMS is an initiative from the European Community to create a unique signaling standard as a cornerstone for the achievement of the interoperability of the trans-European rail network and is likely to be adopted by the rest of the world as well. Ansaldo STS, an industry leader in Europe for the railway system, provides its implementation of the ERTMS/ETCS written in standard ANSI-C, which consists of thousands of code lines. In the field of railway signaling systems the CENELEC EN50128 [3] guidelines for software development recommend the use of formal methods where it is possible and a 100% of testing coverage where it is impossible. In this paper the case study provided by Ansaldo STS has been verified with CBMC obtaining the target 100% code coverage, requested by CENELEC guidelines. The use of our methodology for test generation led to a dramatic increase in the productivity of the entire Software Development process by substantially reducing the number of tests generated and thus the costs of the testing phase. The paper is structured as follows: we firstly describe a simple and naive algorithm for test generation from a set of independent path, making the assumption of not having unfeasible paths in the program. Then we describe how to modify the algorithm in order to release the assumption and we finally show the experimental results and the conclusions.

2 Branch Coverage Via Path Generation

In this section we first present some basic notions used through the paper, then the basic algorithm is presented, under the assumption of not having to test program with unfeasible paths and finally we present the advanced algorithm taking into account also the unfeasible paths.

2.1 Basic Definitions

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

- N is a set of nodes
- E is a binary relation on N (a subset of N x N), referred to a set of edges;
- $n_s \in N$ and $n_e \in N$ are unique entry (start) 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. In most representations 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 *path* p in G is a sequence $p = \{n_1, n_2, ...n_k\}$ of nodes such that $n_1 = n_s$ and $n_k = n_e$ and for all $i, 1 \leq i \leq k$, $(n_i, n_{i+1}) \in E$. 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 it exists an assignment to the program's input X for which the path is traversed during the program execution, otherwise the path is *unfeasible*. Given a set of path of a control flow graph, an *independent path* is a path such that: there is at least an edge in the path that does not occur in any other path in the set. A branch is defined as any decision outcome in the source code. This is reflected in the unwinded flow graph: for every decision we have one or more nodes with two or more outgoing edges, representing the branches. Different edges in the unwinded graph may refer to the same original branch. Covering a branch requires producing at least a test that contains at least an edge which referes to that branch. CBMC is a Bounded Model Checker for software verification [1], that takes as input a low-level C program and it checks safety properties such as the correctness of pointer constructs, array bounds, and user-provided assertions. Given a program D, and a property P, the verification is done translating both the program and the property into a Boolean formula in Conjunctive Normal Form (CNF) and giving the result to a SAT solver like MiniSat [4]. If the SAT solver returns false then the property holds, otherwise the property does not hold. The conversion from a C program into a CNF consists of three steps:

1. Each function call should be replaced by its function body;

2. Each loop should be unwound, i.e. the body has to be duplicated k times, where k is a parameter given in input (*goto* statements are unwound in a similar way). Notice that each copy of the body is guarded by an if statement that uses the same condition of the loop statement.

3. The program and the property are rewritten into an equivalent program in Single Static Assignment (SSA) form [2] that is an Intermediate Representation (IR) where each variable is assigned exactly once. In the original IR existing variables are split into versions, new variables are typically indicated by the original name with a subscript, so that every definition gets its own version.

Given the property P and a program D both in SSA form, the formula $D \wedge \neg P$ is first converted into a bit-vector equation, i.e. each variable is represented by a bit-vector of fixed size, and the operations are converted into bit-vector operations. Finally, it is converted into a propositional formula in Conjunctive Normal Form (CNF), by adding intermediate variables. If applying a SAT solver to the CNF, the equation is satisfiable, then the property does not hold, and an assignment to the variables making the formula true is returned. Starting from this assignment, it is possible to construct an error trace showing where the property does not hold in the program. Otherwise, if the resulting CNF is false the property holds. However, we can not conclude by saying that the program is verified, but we can say that it is verified for a given k. Choosing another $k' > k$ does not ensure that the property holds. In order to use CBMC as a test generator we need to modify the code in input as firstly presented in [7]: the code under test is instrumented with an *assert(0)* after each branch; the instrumented code is given as input to CBMC with an assert activated at the time. For each assert, CBMC returns an assignment to the input variables (a test) violating the property (*assert(0)*). The asserts are inserted in a naive way without any reasoning, this would generate more tests than necessary. Let's take the following example:

if a == 10 **then** B_1
else B_2
if a $\neq$ 5 **then** B_3
else B_4

given the methodology in [7], it will generate four different program codes adding an *assert(0)* at the end of each block, namely B_1, B_2, B_3, B_4. The set of tests

$$\mathrm{T} = \{t_1, t_2, t_3, t_4\}$$

generated covers the 100% of the branches for construction, and for example

```
set of paths pathGenerator(C)
0    P = {};
1    A_u = update(P,C);
2    while |A_u| ≠ 0 do
3      P = P ∪ genApath(n_s,A_u);
4    return P;

path genApath(n,A_u)
5    if n ≡ n_e then return {n}
6    max_n = succ(n).first();
7    foreach s_i ∈ succ(n) do
8      if |A_u(max_s)| < |A_u(s_i)| then
9        max_s = s_i
10   else if |A_u(max_s)| == |A_u(s_i)| then
11     if E(n, max_s) ∉ A_u then
12       max_s = s_i
13   A_u = update(P,C);
14   return {n} ∪
           genApath(max_s,A_u)
```

Figure 2: Generate Paths Functions.

```
set of tests ATGviaCBMC(P,D)
15   T = {};
16   foreach p ∈ P do
17     D' = instrument(D,p)
18     T = T ∪ CBMCcall(D');
19   return T;
```

Figure 3: Test Generation function

they could be:

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 test from the set the 100% of branch are still covered, for example also the test set $T' = \{t_1, t_3\}$ covers the 100% of the branches. In the naive algorithm we always generate more test of the ones needed to grant the 100% of branch coverage. This may increase the costs associated to the testing phase, in particular during the regression phase. In order to reduce the size of the test set we first construct a set of independent paths covering the 100% of the branches of the control flow graph. Then, given a path in the set, it is possible to instrument CBMC in order to generate a test following the path.

2.2 Basic Algorithm

The testing process is presented in figure 1. It assumes that all the paths are feasible. This is a big restriction that is made only for a better comprehension of the algorithm. In the next subsections the algorithm that takes

```
                                    int TEST1(int[ ] a, int size)
                                    s0      int b = 0; int c = 0;
                                    b1      ASSUME(c < size);
                                    b3      ASSUME(a[c] < 0);
                                    s4        b++;
                                    s4        c++;
int TEST1(int[ ] a, int size)       b1      ASSUME(c < size);
s0        int b = 0; int c = 0;     b3      ASSUME(!(a[c] < 0));
b1, b2, s1 for ( ; (c < size); c++) s4        c++;
b3, b4      if (a[c] < 0)           b1      ASSUME(!(c < size));
s2            b++;                          ASSERT();
s3        return b;                 s13     return b;
```

Figure 4: An example function, left, and its instrumentation on the right.

into account unfeasible paths is presented.

The process in figure 1 firstly analyzes the code under test and it constructs a set of independent paths covering the 100% of the branches in the control flow graph associated to the program under test (PATHGENERATOR), i.e. each path covers at least a branch not covered by any other path. Then, from each path, the module ATG via CBMC will generate a test in a way similar to the one presented in [7]. For each path, the original code is instrumented and it is given to CBMC, which will return an assignment to the input variables, a test, soliciting the path through the code. The process is composed by three main functions: PATHGENERATOR, GENAPATH, ATGVIACBMC, as shown in figure 2 and 3.

pathGenerator is a function presented into details in figure 2, left, that takes as input the control flow graph representation of the program under test (C). It returns a set of basis paths (P) covering the 100% of the branches in C. P is initially empty. A_u is the set of branches in C not yet covered by any path in P. A_u initially contains all the branches in C. Given a node n $\in$ C, $A_u(n)$ is a set of branches not yet covered in C, that can be covered from n. Given a node n, $A_u(n)$ is defined recursively as follow:

$A_u(n_e) = \{\}$

$$A_u(n) = \bigcup_{i=1}^{|succ(n)|} \left(A_u(s_i) \cup \begin{cases} \{E(n,s_i)\} & : \quad E(n,s_i) \in A_u \\ \{\} & : \quad E(n,s_i) \notin A_u \end{cases} \right)$$

where n_e is the end node of the control flow graph, s_i is a successor of n, $E(n_1, n_2)$ is the branch from n_1 to n_2 and $succ(n)$ is the set containing all the successors of n. So for example, given a control flow graph C, s_0 a root node and s_1 is a successor of s_0 connected with a branch b_1. Let be s_2 a successor of s_1 connected with a branch b_2, then $A_u(s_0) = \{b_1, b_2\}$. Notice that $A_u = A_u(n_s)$. Looking at the algorithm in figure 2, left, that

```
int TEST1(int[ ] a, int size)
s0          int b = 0; int c = 0;
s1,b1,b2    if (c < size)
s2,b3,b4      if (a[c] < 0)
s3              b++;
s4            c++;
s5,b1,b2      if (c < size)
s6,b3,b4        if (a[c] < 0)
s7                b++;
s8              c++;
s9,b1,b2        if (c < size)
s10,b3,b4         if (a[c] < 0)
s11                 b++;
s12               c++;
s13         return b;
```

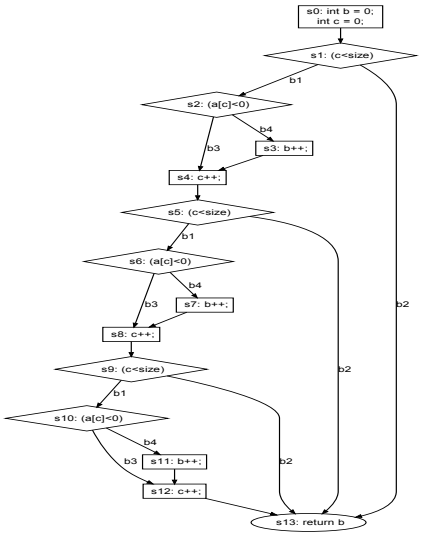

Figure 5: The example code and the corresponding flow graph

generates a set of paths, it starts from an empty set of paths (line 0) and a set of uncovered branches containing all the branches of C (line 1). The function UPDATE(P,C) returns all the branches of C not yet covered by P. Moreover for each node n in C, it will update $A_u(n)$. Then, while there are still uncovered branches in C(line 2), a new path is generated and added to P (line 3).

genApath is a recursive function that given a set of uncovered branches (A_u) and a starting node (n) generates a path containing at least an uncovered branch in A_u. It always starts from the beginning of the control flow graph (line 3) from the node n_s. The idea is: starting from a node n, the algorithm explores successors of n, s_i, and it chooses the one having the greatest number of uncovered branches (from line 8 till 12). If a tie is found between two successors of n (line 10), n_1 and n_2, then the one having $E(n, n_i) \in A_u$, in lexicographic order, is chosen. The algorithm stops whenever the end of the control flow graph (n_e) is reached line 5. Then the path is constructed in a backtrack way, starting from the last node n_e, including all the nodes explored by the algorithm (line 14). The algorithms in figure 2 left and right only generate a set of independent paths.

ATGviaCBMC is a function that takes as input a set of paths and it will return (if exists) a test for the program under test representing each path. T is the set of tests generated, initially empty (line 15). Then for each

path the program D is modified, i.e. opportunely instrumented,(line 17) and a new program, D' is created. D' is then passed to the tool CBMC(line 18) that returns as assignment to the input variables exploring the path in input. D' has to be carefully instrumented in order to generate a test. In [7] an *assert(0)* is added when a branch needs to be covered. However CBMC chooses which path has to follow in order to generate a test to cover the assert. We can still use CBMC but we need something to force CBMC to reach an assertion through a 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 at the time in the code, obtaining from CBMC a test exploring the path. In order to obtain the 100% of the branch coverage for each independent path we produce an instrumented code using the *assume* construct to enforce the path and we put an *assert(0)* at the end of the code. Then for each instrumented code we run CBMC that returns a test for that path.

An example of the algorithm behavior is presented below. Let's assume as a function under test the function presented in figure 4 which counts negatives in an array. The algorithm presented above takes as input C, i.e. the control flow graph representation of the function in figure 4, left already unwinded. The set of paths, line 0, is initially empty and the A_u is updated (line 1) with all the branches of the Control Flow graph, $A_u = \{b_1, b_2, b_3, b_4\}$. For each node n it is also updated $A_u(n)$. Notice that the function GENAPATH will work on the control flow graph unwinded, which is shown in figure 5, right, assuming an unwind value of 3. Each function has a minimum unwind value k, necessary to reach full coverage, which depends on the cycles structure. Best way to set k would be to know such minimum a priori but that's not feasible in an automatic way so, a good alternative solution is to start testing with low k and increase it until coverage is reached. Lower k are faster to test but have less chances to reach full coverage while bigger k requires more time and effort for both CBMC and our algorithm and also increase the chances that CBMC will fail to find an answer in the given time. At the first node there is only one branch (without any label) and then the recursive call to GENAPATH(s_1,A_u) is executed. At this point since s_1 is not the end node, its successor nodes (s_2 and s_{13}) would be explored. $max_s = s_2$, since $|A_u(s_{13})| = 0$ and $|A_u(s_2)| = 4$ ($\{b_1, b_2, b_3, b_4\}$). The current coverage status is then updated by deleting b_1, which is the incoming edge used to reach s_2, from both A_u and from each $A_u(n)$ where n is a node in C. A new recursive call, GENAPATH(s_2,A_u) is executed. Then the algorithm chooses between s_3 and s_4, having $|A_u(s_3)| = |A_u(s_4)| = 3$, ($\{b_2, b_3, b_4\}$) and being both edges b_3 and b_4 uncovered, the algorithm breaks the ties by lexicographic order choosing s_3. A_u is updated deleting b_4 and it is also updated for each node. From node s_3 to node s_5 there is a single path to be followed.

On node s_5 $max_s = s_6$ with $|A_u(s_6)| = 2$ ($\{b_2, b_3\}$) against $|A_u(s_{13})| = 0$. From s_6, s_8 will be chosen because even if $|A_u(s_7)| = |A_u(s_8)| = 2$, b_3 is not yet covered. Finally the algorithm will go from s_9 to s_{13}, covering b_2. As soon as the algorithm reaches the s_{13} node it will stop and backtrack until the root constructing the path, i.e. $p_1 = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_8, s_9, s_{13}\}$. This path can cover all the branches, so no more calls will be necessary. The pathGenerator function will return a set of paths, $P = \{p_1\}$ and then the function ATGviaCBMC is called. The function instrument the code in order to force CBMC to follow the path. In figure 4, right, the modified code (C') for the generated path is presented. Notice that:

- the instrumented code is already unwinded,
- 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, will generate a test, i.e. with the following inputs, $size = 2$, $a[0] = -1$ and $a[1] = 0$.

2.3 Using CBMC for Unfeasible Path

In case of unfeasible paths the algorithm presented above will generate a set of tests that does not cover the 100% of the branches. In order to overcome the problem whenever a path is generated, a checker is used to analyze the path and return if the path is feasible or not. In case of unfeasible path a new path has to be generated. As feasibility checker we use CBMC: if a test is returned from CBMC then the path is feasible otherwise is not. The new algorithm is composed by a function CreateTestSet that uses the function generatePath, that it is similar to the one presented above: this function also takes as input a set of branches that have not to be covered from the path returned. In case that this set is empty the function is equivalent to the one presented above.

CreateTestSet is a function presented in figure 6 that given a set of branches(A_u) and a program P returns a set of tests covering the 100% of the branches in A_u. Given a finite set of paths of a program P (P^k), generate a path covering as much branches in A_u as possible CreateTestSet generates a path covering as much branches as possible of A_u (line 2) starting from the basic path (b_p) and avoiding the Forbidden branches (F). If such path exists, then CBMC is invoked (line 4) and if the path is feasible CBMC returns a vector of values for the input variables; then the test is added to the set of Tests T (line 8) and the branches traversed by P are eliminated from A_u. If the path is unfeasible the function analisePath returns (if any) the subset of arch which make it feasible (b_p) and the arches

```
set of Tests createTestSet(A_u, P, k_max)
0  k = 4; T = {}; b_p = {n_s}; F = {}
1  while |A_u| > 0 & k < k_max do
2    p = generatePath(A_u, b_p, F, P^k)
3    if |p| > 0 then
4      t = generateTest(p, P^k)
5      if t == t_0 then
6        ⟨b_p, F⟩ = analisePath(b_p, p, F, P^k)
7      else if t == t_stop then
8        return T
9      else
10       T = T ⋃ {t}
11       b_p = {n_s}
12       A_u = update(A_u,p)
13   else
14     if b_p ≠ {n_s} then
15       F = {b_p}
16       b_p = b_p \ tail(b_p)
17     else
18       k=k*2; F ={}; b_p = {n_s}
19 return T
```

Figure 6: Advanced algorithm taking into account unfeasible paths

making the path unfeasible are stored in F (line 6) so that the next path will be generated starting from b_p and avoiding the forbidden path. If the path generated in line 2 is empty, i.e. does not exists any path starting from b_p and covering at least a branch in A_u such that the branch is forbidden, then a new basic path is computed, i.e. the tail of the basic path is removed and it became a forbidden arch. If b_p was the root, $|A_u|$ is not equal to zero, then there are some branches not covered with the fixed k: at this point (line 16) k is incremented and the search restarts again. The function returns the set of tests so computed.

Suppose, for example, that the function in figure 4 did not have the size of the array as an input, but instead it is hard coded in the cycle and it is greater than 2, i.e. 3. Also necessarily consider a greater unwind value, i.e. 4 instead of 3. In this case the previous algorithm would generate the exact same path but that would be unfeasible because it tries to exit from the cycle after two iterations, which cannot be done. The new algorithm will then exclude that possibility by removing and forbidding the last step from the path and compute another path with the same initial subpath which will stay in the cycle one more iteration and exit afterwards. This will be found feasible and reported as a test.

module	functions	naive	manual	atg
Module_01	26	154	127	106
Module_02	12	149	57	43
Module_03	1	12	17	6
Module_04	1	83	43	41
Module_05	2	76	18	7
Module_06	8	123	39	27
Module_07	8	44	27	18
Module_08	24	280	200	148
Module_09	16	182	144	108
Module_10	9	98	35	38
Module_11	22	340	312	263
Module_12	12	35	34	32
Module_13	11	210	181	156
Module_14	11	196	157	132
Module_15	29	335	322	238
Module_16	14	96	69	29
Module_17	7	73	36	28
Module_18	26	170	101	82
Module_19	8	89	62	38
Module_20	25	161	119	110
	272	2906	2100	1650

Figure 7: Modules with full coverage.

3 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 [5] 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:

- ETCS, the European Train Control System, transmits speed information to the train driver and it monitors constantly the driver's compliance with the speed information;
- GSM-R is based on standard GSM but it uses various frequencies specific to rail as well as certain advanced functions. It is the radio system used to exchange voice and data information between the track and the train.

module	functions	unreachable path	time-out
Module_21	11	1	0
Module_22	9	2	0
Module_23	13	2	0
Module_24	19	1	0
Module_25	18	1	0
Module_26	3	1	0
Module_27	23	1	0
Module_28	21	2	3
Module_29	7	0	2
Module_30	9	2	2
Module_31	10	0	4
	143	13	11

Figure 8: Modules with partial coverage.

Ansaldo STS as part of the 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 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 seen in the section above. We get more than 130 different modules, containing more than 100.000 lines of code distributed in more than 1700 functions. As a comparison we only took 31 different modules, the modules on which we get informations about the tests manually generated by Ansaldo STS. As unwind, after a brief analysis, we start from k = 5 till k = 100, with step 2. We also set a 20 minutes timeout for both our algorithm and CBMC.

In figure 7 are reported the results on the module on which we reach the 100% of branch coverage. The columns from left to right represent: the name of the module (for copyright reasons they are coded), the number of functions contained in the module, the number of tests needed to reach the 100% of branch coverage using the naive method presented in [7], the number of tests using manual generation, provided by Ansaldo STS, the number of tests generated by our technique presented above. Our algorithm on 20 modules always reach the 100% of the branch coverage. Firstly notice that the number of tests generated by the naive method is always greater than the number of tests generated manually: In 5 cases the number of tests manually generated by the naive algorithm is more than a factor two with

respect the number of tests automatically generated, while there is only one case where manual generation creates more tests than the naive method, i.e. for the $module_{03}$. This is mainly due to the manual generation method. For each function Ansaldo STS maintains a set of functionality tests say T_f: if T_f already guarantee branch coverage then no more test are generated, otherwise a new test set T_c is generated to cover remaining branches. The final test set for coverage will be: $T = T_f \cup T_c$, which frequently contains redundant test. In this case, for the $module_{03}$, the number of tests for functionality covers the 100% of the branches and from a coverage point of view, there are some test that are redundant. Moreover, looking at figure 7, we always automatically generate a test set having size smaller that the one generated by Ansaldo STS. Only $module_{10}$ does not follow this rule mainly due to the unwind strategy used: starting from greater k, it is possible to generate smaller test set covering the 100% of the branch [6]. As a matter of fact, for all the other modules, the number of tests is always less than the number of tests manually generated. The time used to generate the tests is very low and it is not comparable with the time spent by Ansaldo STS to manually generate them, as already described in [7].

In figure 8 are also reported the modules on which we were not able to obtain full coverage: the first column reports the name, the second column the number of functions in the module, third the number of functions not covered due to unreachable code and the last column reports the number of functions not fully covered due to CBMC time-out. The functions that are not reported in the last two columns, have been fully covered, with less test than manual generation. As shown in the table, few functions have been reported as uncoverable, in particular the functions presenting unreachable codes have been manually checked. Also Ansaldo STS has reported that these functions can not be fully covered. The results, as shown in the tables, are that only 4 modules out of 31 can not be completely covered (13% of the modules) and the number of functions that can not be covered are 11 (resp. 13) for timeout (resp. unreachable code) out of 415.

4 Conclusion

In this paper we have presented a methodology for the automatic generation of test set for Coverage Analysis, containing only non redundant test. The generation goes through the construction of a set of feasible (if exist) independent paths. We have experimented our methodology on a subset of 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 were able to verify completely 20 out of 31 different modules of the ERTMS. On the remaining we were able to conclude that 7 out of 31 could not be covered due to unreachable code while for the last 4 out of 31 has

been impossible to reach complete coverage since CBMC could not finish. This paper has shown that the use of our methodology for test generation led to a dramatic increase in the productivity of the entire Software Development process by substantially reducing the number of tests generated and thus the costs of the testing phase.

References

[1] E. Clarke, D. Kroening, F. Lerda. : A Tool for Checking ANSI-C Program. Tools and Algorithms for the Construction and Analysis of Systems, 2004, pp. 168–176.

[2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck. : Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In ACM Transactions on Programming Languages and Systems, 1991, vol. 13, number 4, pp. 451–490.

[3] European Committee for Electrotechnical Standardization. : Railway Applications - Communication, signalling and processing systems - Software for railway control and protection systems. http::://www.cenelec.eu

[4] N. Een, N. Sorensson. : An Extensible SAT-solver. In Satisfiability Workshop, 2003, pp. 502-518.

[5] ERTMS: The official Website. http://www.ertms.com/, 2008.

[6] A. Arcuri. : Longer is Better: On the Role of Test Sequence Length in Software Testing. 2009

[7] D. Angeletti, E. Giunchiglia, M. Narizzano, A. Puddu, S. Sabina. : Automatic Test Generation for Coverage Analysis of ERTMS software. International Conference on Software Testing, Verification, and Validation (ICST). Denver (CO) 2009.

[8] B. Beizer. : Software testing techniques. New York: Van Nostrand Reinhold Co. 1990.

[9] H. Chockler, O. Kupferman, R.P. Kurshan, M.Y. Vardi. : A Practical Approach to Coverage in Model Checking. n Proceedings of the 13th international Conference on Computer Aided Verification, 2001, vol. 2102. pp. 66-78.

[10] J. de Halleux and N. Tillmann. : Parameterized unit testing with pex.

[11] Luke Gregory: Path Testing. 2006.