# Automatic Test Generation for Coverage Analysis of ERTMS software [*]

Damiano Angeletti[‡] Enrico Giunchiglia[†] Massimo Narizzano[†] Alessandra Puddu[†]
Salvatore Sabina [‡]

## Abstract

*ERTMS is the European Railway Traffic Management System. The CENELEC EN50128 guidelines for software development of safety critical system require that the software produced is verified by providing a set of tests covering the 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 this paper we present a methodology to automatic generate test achieving the desired code coverage. The automatization of the test generation phase, applied to some modules of the ERTMS developed by Ansaldo STS (an Italian leading company in the field), led to a dramatic increase in the productivity and to a reduction of the costs of the entire Software Development process.*

## 1 Introduction

The importance of software verification, i.e. ensuring that a developed software does not contain errors, is well known, in particular in the field of safety-critical systems where, a failure, could have catastrophic consequences. The most used technique for software verification remains testing: 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 CENELEC EN50128 [2] guidelines for software development of safety critical system require that the software produced is verified by providing a set of tests covering the 100% of the code. This requirement, however, substantially increases the costs associated to testing: given a set of tests $T$ (either automatically or manually generated), if $T$ fails to cover a portion $P$ of the system under test, a new set of tests $T'$ has to be devised in order to cover $P$, and usually these new tests are manually generated.

In this paper we present a methodology to automatic generate test achieving the desired code coverage. As main core we use CBMC [3] a Bounded Model Checker that, applied to low-level ANSI-C programs, it checks safety properties such as the correctness of pointer constructs, array bounds, and user-provided assertions. In particular, provided an user assertion, CBMC will generate an error trace, violating the assertion from which we start to construct our test set. The automatization of the test generation phase, applied to some modules of the ERTMS [4] developed by Ansaldo STS (an Italian leading company in the field), led to a dramatic increase in the productivity and to a reduction of the costs of the entire Software Development process. In the following we start showing the industrial setting used to generate tests for coverage analysis of the ERTMS. Then we present our new methodology for the automatic test generation using CBMC. Finally we will presents some experimental results and possible directions of this research.

## 2 Industrial Setting

Today, trains are equipped with up to six different navigational systems where each one is extremely costly and takes up space on-board. A train crossing from one European country to another must switch the operating standards as it crosses the border, adding to travel time and operational and maintenance costs.

The European Rail Traffic Management System (ERTMS) is an EU "major European industrial project" to enhance cross-border interoperability and signalling procurement by creating a single Europe-wide standard for railway signalling.

ERTMS has two basic components: ETCS, used for exchanging data information between the train and the driver, and GSM-R, based on standard GSM, used for exchanging data information between the track and the train.

Ansaldo STS offers its implementation of the ERTSM system, and in particular it produces the European Vital Computer (EVC) software, a fail-safe system which super-

[*]Partially supported by a Ph.D. grant (2007-2009) financed by Ansaldo STS

[†]DIST, Università di Genova, Viale Causa 13, 16145 Genova, Italy, {name.surname}@unige.it
[‡]Ansaldo STS, Via Paolo Mantovani, 3 - 16151 Genova Italy, {surname.name}@asf.ansaldo.it

```
int FUT(int a)
s_0      int r = i = 0
b_0,b_1  while i < max do
s_1        g++
b_2        if i > 0 then
s_2          a++
b_4,b_5      if a ≠ 0 then
s_3            r = r + (g+2)/a
b_3        else
s_4          r = r + g + i
s_5        i++
s_6      r = r * 2
s_7      return r
```

```
#ifdef ASSERT_i
assert(0)
# endif

int FUT(int a)
         ASSERT_1
s_0      int r = i = 0
b_0,b_1  while i < max do
         ASSERT_2
s_1        g++
b_2        if i > 0 then
         ASSERT_3
s_2          a++
b_4          if a ≠ 0 then
         ASSERT_4
s_3            r = r + (g+2)/a
b_5          else
         ASSERT_5
b_3        else
         ASSERT_6
s_4          r = r + g + i
s_5        i++
         ASSERT_7
s_6      r = r * 2
s_7      return r
```

**Figure 1. An example of function and its instrumentation**

vises and controls the speed profiles using the information received from the track. The CENELEC EN50128 is a part of a group of European standards for the development of Railway applications. It concentrates on the methods which need to be used in order to provide software which meets the demands for safety integrity. The European standards have identified techniques and measures for 5 levels of software safety integrity ($sil$) where 0 is the minimum level and 4 the highest level. The railways system requires a $sil$ 4, and the system verification should be done producing a set of test having 100% of code coverage. With a test we mean an assignment to the input variables of the function under test and with code coverage we mean *branch coverage*, i.e. the percentage of branching condition executed by the program under a set of test [7]. In order to reach the requirement most of the tests in Ansaldo STS are generated manually and the entire process can be summarized in two main steps:

- *testing generation*: starting from a set of test (initially empty) a new test ensuring that a given part of the code is covered, is created, i.e. a value for the input variables is setted.
- *branch coverage computation*: a tool Cantata, see [5], is run on the function under test using the set of test constructed, returning the percentage of branch covered by the set of tests. If the 100% of branching coverage is reached, then the testing generation phase will end. Otherwise a new test is generated, and as a guide for the generation of the next test, Cantata will report also which branches are not covered by the set.

As an example take the code in Figure 1 left, where each statement is marked with an $s_i$ label and each branch, and its negation, is marked with an $b_i$ label. Given an empty set of test, we can construct a test that will explore the branches $\{b_0, b_3, b_1\}$, choosing for example $t(g, max, a) = \langle 0, 1, 0\rangle$, where {g,max,a} is the set of input variables: it covers almost the 42% of the entire branches, and thus a new test has to be generated. This is a very expensive process and Ansaldo STS estimates that for each test generated a person will spend *fifteen minutes* in average.

## 2.1 Automatic test generation for Coverage Analysis

The Software Verification process consists into verifying that a program always satisfy a property $P$. In *SAT-based Bounded Model Checking* given a program, and a property $P$, the verification is done translating both the program and the property into a a boolean formula in Conjunctive Normal Form (CNF) and giving the resulting equation to a SAT solver like chaff [6]. If the SAT solver returns true then the

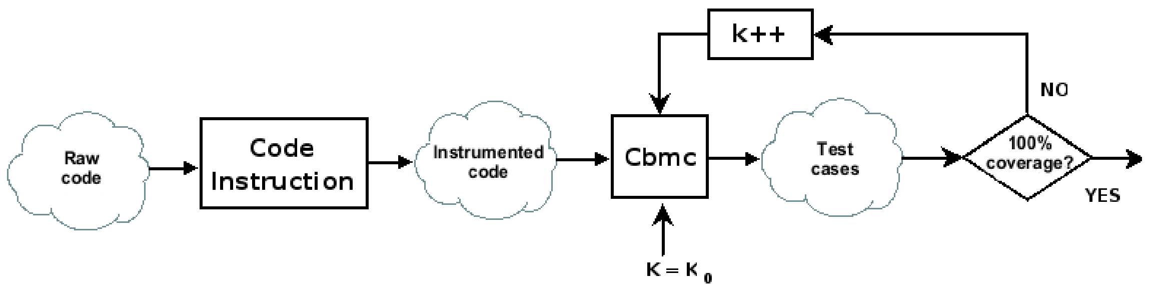

**Figure 2. Testing Process with CBMC**

property is violated and an assignment to the variable violating the property is returned. Otherwise if the resulting CNF is false, then the property holds. Notice that BMC techniques usually deal with problems with a fixed bound on the loops, so given a program with unbounded loops, a bound $k$, and a property to verify, if the property holds for the given $k$, we can not conclude that the property will never be violated, and in this case we should check its violation with a greater $k$, until either the property is violated or no more increment of $k$ are possible. BMC for Software Verification was first proposed with success in a tool called CBMC: a Bounded Model Checker for C programs [3]. CBMC takes as input a low-level C program and it allows the check of safety properties such as the correctness of pointer constructs, array bounds, and user-provided assertions. Among the other features of CBMC, it allows the possibility to check if a part of code is never reachable given an assignment to the input variables. This is done adding the property $assert(0)$ to the line that we want to prove that is not unreachable: CBMC will return an error-trace if the property is violated, i.e. if the line with the assertion is reached. Using this feature we can generate a test that reach a particular statement $s_i$ in the code, just adding an $assert(0)$ before $s_i$ and running CBMC on it. So in principle we can think to add to a program an $assert(0)$ for each branch instruction and its negation and running CBMC for each assertion. In this way we can construct a set of test covering the 100% of branch instruction of the program. In Figure 2 is presented the methodology for the use of CBMC as an automatic test generator, and it is composed by three main steps:

1. *Code Instrumentation* : CBMC requires that each function called in the function under test is completely defined. However a tester can not always have the access to each function, for example if they are defined in modules not available. So for each missing function has to be created a function $stub$, i.e. a function returning a random value. The stubs can be either manually or automatically inserted. Moreover it is also necessary for CBMC the creation of a function $main$ calling the function under test, and setting all the value of the input variables. Finally in CBMC we need to insert an $assert(0)$ for each branch predicate and one for its negation. Unfortunately they can not been inserted all together since during the generation phase CBMS stops at the first assertion violated. An alternative could be insert an assert at the time, generating one file for each assertion introduced. However, in this way many files will be generated, one for each assertion, causing difficulties in the management of the test cases. So if we want to generate $n$ tests we have to introduce $n$ macros like:
   #if defined(ASSERT_i)
   assert(0)
   #endif
   where $i = 1..n$. Then, instead of insert assert(0) in the function under test, we introduce an ASSERT_i. In Figure 1 left, is presented the function under test and on the right the code instrumented for CBMC usage, where each ASSERT_i represents a test that has to be generated. In order to generate a test for $b_5$ an else block with only the ASSERT_5 macro has been added.

2. *Test Generation*. After the instrumentation, CBMC is run $n$ time with a fixed k, where $n$ is the number of assertion added to the code, and for each run an assertion is activated. From each run a counterexample is produced from which is extracted a test.

3. *Coverage Analysis*. Adding an assertion for each branch does not ensure 100% of branch coverage since we are running with a fixed k. For the example in Figure 1 the statements $s_2$, $s_3$ and $s_4$ can not be covered with $k = 0$. So, given a fixed k, if a branch is not covered then k is increased and CBMC is run again until either k can not be increased anymore or 100% of branch coverage is reached. For Coverage Analysis we use Cantata. Sometimes due to the problem hardness CBMC can not terminate in a reasonable amount of time. In this case, the manual generation should be used. In case of ”unreachable code ” CBMC can either not terminate or can reach the maximum k with-

**Table 1. Experimental Evaluation on ERTMS**

| | | Cbmc | | Ansaldo STS | |
|---|---|---|---|---|---|
| | # function | # test | time(s) | # test | time(s) |
| $mod_1$ | 19 | 148 | 1212 | 64 | 57600 |
| $mod_2$ | 7 | 47 | 1444 | 26 | 23400 |
| $mod_3$ | 13 | 193 | 6256 | 80 | 72000 |
| **Total** | **39** | **388** | **8912** | **170** | **153000** |

out generating a test. In the first case the check is left for the tester, in the second case it can be stated that the code in unreachable.

## 3 Experimental Analysis

In Table 1 we show a preliminary experimental analysis on three different module of the ERTMS implemented by Ansaldo STS. For our experimental analysis we used a normal desktop pc, with 1 GB of memory, running Linux Ubuntu Gutsy Gibbon as Operating System. As an underline the entire process has been also run under Windows system. In the first column, for industrial copyright purpose, we omitted the names of the modules under test, substituting them with $mod_i$, where i is a number between 1 to 3. The second column represents the number of different function in the module, while the third and fourth columns the number of test automatically generated by CBMC and the time needed to generate them respectively. In the fifth and sixth column we put the number of test generated by Ansaldo STS and the expected time used to generate it. The time in the sixth column has been calculated multiplying the number of tests in column # test of the Ansaldo STS side, with fifteen minutes that it is the average time calculated by Ansaldo STS for generating a test. All the test generated have been proved to cover the functions under test with the 100% of branch coverage. As can be seen from the table the automatic test generation can reduce the time of more than 2 order of magnitude (looking at the row total and the columns time(s)). This is also true for each module tested, and also if can not been shown from the figure it is also true for each function. Moreover the automatic approach always produces a set of test greater than the one necessary for the coverage, as testified by the tests generated by Ansaldo STS. However, even if the set of tests is greater the time spent for generating it is still smaller. Notice that a greater test set can increase the regression phase costs. However, in the Ansaldo STS environment these costs were small and we did not take them into account. The more tests generated is mainly due to naive implementation of our methodology that, for example, may generate a test that cover a subset of branches covered by the tests already generated.

## 4 Conclusion and Future Work

In this paper we have presented a methodology for the automatic test generation for software verification of some modules of the ERTMS implemented by Ansalso STS. The methodology proposed is completely automatic with respect to the methodology used in industry. The experimental analysis have shown that the automatic test generation using CBMC can produce speedup more than two order of magnitude, making a significative improvement in the testing process. However, our approach is far to be completed, and there is space for improvements. For example the code generation phase may be modified in order to generate smaller set of tests or to include other kind of code coverage techniques as for example path coverage [1] or Modified Condition/Decision Coverage [8].

## References

[1] B. Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[2] CENELEC. Cenelec 50128: Railway applications software for railway control and protection systems.

[3] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[4] ERTMS. The official website. http://www.ertms.com/.

[5] IPL. Cantata ++. http://www.iplbath.com.

[6] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.

[7] M. Roper. *Software Testing*. McGraw-Hill, Inc., New York, NY, USA, 1995.

[8] S. A. Vilkomir and J. P. Bowen. Reinforced condition/decision coverage (rc/dc): A new criterion for software testing. In *ZB2002: Formal Specification and Development in Z and B, Proceedings of 2nd International Conference of B and Z Users*, pages 295–313. Springer-Verlag, 2002.