J Autom Reasoning (2010) 45:397–414
DOI 10.1007/s10817-010-9172-3

Using Bounded Model Checking for Coverage Analysis of Safety-Critical Software in an Industrial Setting

Damiano Angeletti · Enrico Giunchiglia · Massimo Narizzano · Alessandra Puddu · Salvatore Sabina

Received: 12 October 2008 / Accepted: 30 March 2010 / Published online: 29 April 2010
© Springer Science+Business Media B.V. 2010

Abstract Testing and Bounded Model Checking (BMC) are two techniques used in Software Verification for bug-hunting. They are expression of two different philosophies: testing is used on the compiled code and it is more suited to find errors in common behaviors, while BMC is used on the source code to find errors in uncommon behaviors of the system. Nowadays, testing is by far the most used technique for software verification in industry: it is easy to use and even when no error is found, it can release a set of tests certifying the (partial) correctness of the compiled system. In the case of safety critical software, 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 often involves the manual generation of tests. In this paper we show how BMC can be productively applied to the Software Verification process in industry. In particular, we show how to productively use a Bounded Model Checker for C programs (CBMC) as an automatic test generator for the Coverage Analysis of Safety Critical Software.

Partially supported by a Ph.D. grant (2007–2009) financed by Ansaldo STS.

D. Angeletti · S. Sabina
Ansaldo STS, Via Paolo Mantovani, 3-16151 Genova, Italy

D. Angeletti
e-mail: damiano.angeletti@ansaldo-sts.com

S. Sabina
e-mail: salvatore.sabina@ansaldo-sts.com

E. Giunchiglia · M. Narizzano (✉) · A. Puddu
DIST, Università di Genova, Viale Causa, 13-16145 Genova, Italy
e-mail: massimo.narizzano@unige.it

E. Giunchiglia
e-mail: enrico.giunchiglia@unige.it

A. Puddu
e-mail: alessandra.puddu@unige.it

In particular, we experimented CBMC on a subset of the modules of the European Train Control System (ETCS) of the European Rail Traffic Management System (ERTMS) source code, an industrial system for the control of the traffic railway, provided by Ansaldo STS. The Code of the ERTMS/ETCS, with thousands of lines, has been used as trial application with CBMC obtaining a set of tests satisfying the target 100% code coverage, requested by the CENELEC EN50128 guidelines for software development of safety critical systems. The use of CBMC for test generation led to a dramatic increase in the productivity of the entire Software Development process by substantially reducing the costs of the testing phase. To the best of our knowledge, this is the first time that BMC techniques have been used in an industrial setting for automatically generating tests achieving full coverage of Safety-Critical Software. The positive results demonstrate the maturity of Bounded Model Checking techniques for automatic test generation in industry.

Keywords Automatic test generation · Testing · Bounded model checking

1 Introduction

The importance of software verification, i.e. ensuring that a developed software does not contain errors, is well known, in particularly in the field of safety-critical systems where a failure can have catastrophic consequences. Testing and Bounded Model Checking (BMC), are two techniques used in Software Verification for bug-hunting. They are expression of two different philosophies: testing is done on the compiled code and it is more suited to find errors in common behaviors, while BMC is done on the source code and it is more suited to find errors in uncommon behaviors of the system. BMC has been successfully used in the last decade to formally verify finite state systems, such as sequential circuits and protocols, see e.g. [2]. It was originally proposed by [3] as a complementary technique to OBDD-based model checking. The key idea of BMC is to first build a propositional formula whose models correspond to program traces (of bounded length) that violate some given properties, and then check the resulting formula for satisfiability. In [10] it has been showed how this technique can be productively used in industry and since then many hardware companies, like Intel and IBM, started to introduce BMC in the quality assurance process. Recently BMC has been also applied to the Software Verification. In particular, in [7] the authors built a robust Bounded Model Checker, called CBMC, for ANSI-C and C++ programs. CBMC allows verifying array bounds, pointer safety, exceptions and user-specified assertions. Despite these works, testing remains by far the most used technique for software verification in industry: it is easy to use and even when no error is found, it can release a set of tests certifying the (partial) correctness of the compiled system. In the case of safety critical software, in order to increase the confidence in the correctness of the compiled system, it is often required that the set of tests covers 100% of the code. This requirement, however, can substantially increase 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, often, these new tests are manually generated by domain experts.

In this paper we show how BMC can be successfully applied to the Software Verification process in industry. In particular, we experimented CBMC on some of the modules of the ERTMS/ETCS [18] source code 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. Ansaldo STS, an industry leader in Europe in the railway sector, provided its implementation of the ERTMS/ETCS, which consists of thousands of standard ANSI-C lines of code. In the field of railway signalling systems, the CENELEC EN50128 [15] guidelines for software development recommend the use of formal methods if possible, and testing with a 100% of coverage otherwise. In this paper we used CBMC for the automatic generation of a test set obtaining the target 100% code coverage, requested by the CENELEC guidelines. To the best of our knowledge, this is the first time that CBMC has been used for automatic test generation and coverage analysis of Safety-Critical Software in an industrial setting. By comparing with the efforts associated to the manual generation of the tests, we show that the use of CBMC leads to a dramatic increase in the productivity of the entire Software Development process by substantially reducing the efforts associated to the testing phase. These results demonstrate the maturity of the Bounded Model Checking technique for automatic test generation in industry. The paper is structured as follows. First, in Section 2, we review the basics of BMC in Software Verification: this will give us the opportunity to present also the main ideas behind CBMC. Then, in Section 3, we show how it is possible to use CBMC also for automatically generating tests, while in Section 4 we describe (1) Ansaldo STS methodology based on the manual construction and solution of a set of constraints, (2) our proposal to automatize the process based on CBMC, and (3) the results of the evaluation of our proposal on the ERTMS/ETCS case study. We end the paper with some conclusions and the related work (Section 5).

2 Bounded Model Checking for Software Verification

For sake of simplicity, we consider programs written in the subset of ANSI-C language consisting of *if-statements*, *assignments*, *assertions*, *labels* and *goto* statements. We do not loose in generality making this restriction, since it is always possible to convert an ANSI-C program into an equivalent one having only these statements [7]. Figure 1 left, shows an example of code satisfying the restriction, where a, max and g are *input variables*, i.e. variables appearing as input parameters or variables that are not defined but used in the body of the program/function under test.

The Software Verification process consists of verifying whether the given program satisfies a given property. In *SAT-based Bounded Model Checking* [3] given a program, the property, and an additional parameter k representing the *bound*, 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 chaff [24] or MiniSat [17]: if the SAT solver determines that the formula is unsatisfiable then the property holds for the given bound (which can then be incremented), otherwise the property does not hold and a counterexample (extracted from the model computed by the SAT solver) is returned.

The conversion from a C program into a CNF consists of three steps:

1. Each function call is replaced by its body;
2. Each loop is unwound, i.e. the body has to be duplicated k times, where k is the bound (*goto* 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. Figure 1 center, shows the unwound code for $k = 1$ of the code on the left: the body of the *while* (lines 2–9 left) is replicated once (lines 2–9 center), the *while* statement is deleted and an *if* statement is added as a guard (line 1 center).
3. The program and the property are rewritten into an equivalent program in *Single Static Assignment (SSA)* form [11], which is an intermediate representation where each variable is assigned exactly once. In such intermediate representation, each variable x in the original program is split into versions—indicated by the original name with a subscript, e.g., $x_1, x_2, \ldots$—each representing a possible value of x during the execution of the program. For instance, looking at Fig. 1 center,

   (a) We replace the occurrence of r at line 0 with r_0, and similarly for the definitions of r at lines 6 and 8; and
   (b) The only not trivial task is to determine which among r_0, r_1 and r_2 should be used in place of r at the right hand side of line 10: to this end, variables r_3 and r_4 are added and properly set.

Figure 1 right, represents the SSA form of the program in Fig. 1 left, with $k = 1$. If we interpret the assignment symbol in the SSA as equality, we obtain an equational formula corresponding to the original program with $k = 1$. Given the formula C corresponding to the program (for a given k) and a formula P in the language of C representing the property to verify, the formula $C \wedge \neg P$ is

1. First converted into a propositional one by representing each variable as a bit-vector of fixed size and the operations as bit-vector operations (see [8] for details),

```
int FUT(int a)
0  int r = i = 0
1  while i < max do
2     g++
3     if i > 0 then
4        a++
5        assert(a ≠ 0)
6        r = r + (g+2)/a
7     else
8        r = r + g + i
9     i++
10 r = r * 2
11 return r
```

```
int FUT(int a)
0  int r = i = 0
1  if i < max then
2     g++
3     if i > 0 then
4        a++
5        assert(a ≠ 0)
6        r = r + (g+2)/a
7     else
8        r = r + g + i
9     i++
10 r = r * 2
```

$$
\begin{aligned}
C := \ & r_0 = 0 \wedge \\
& i_0 = 0 \wedge \\
& g_1 = g_0 + 1 \wedge \\
& a_1 = a_0 + 1 \wedge \\
& r_1 = r_0 + \tfrac{g_1+2}{a_1} \wedge \\
& r_2 = r_0 + g_0 + i_0 \wedge \\
& i_1 = i_0 + 1 \wedge \\
& r_3 = i_0 > 0 ? r_1 : r_2 \wedge \\
& r_4 = i_0 < max_0 ? r_3 : r_0 \wedge \\
& r_5 = r_4 * 2 \wedge \\
P := \ & a_1 \neq 0
\end{aligned}
$$

Fig. 1 Example of SSA transformation; *Left*: A generic function written in a subset of the ANSI-C; *Center*: Unwinding with $k = 1$; *Right*: SSA form transformation of the program on the *left*

2. And then into Conjunctive Normal Form (CNF) using well known conversion methods (see, e.g., [26]).

BMC for Software Verification, as described above, was first successfully proposed in a tool called CBMC: a Bounded Model Checker for C programs [7]. CBMC takes as input a C program and it allows to check safety properties such as the correctness of pointer constructs, array bounds, and user-provided assertions. In order to show how CBMC works, consider the example in Fig. 1. Fixing $k = 1$ and running CBMC, first the program in Fig. 1 center, and then the formula in Fig. 1 right are produced. Assuming that the property to verify is that a division by zero will never occur—represented by the *assert(a* $\neq$ *0)* at line 5 in Fig. 1 left—CBMC will determine that the property holds for $k = 1$. However, for $k = 2$, CBMC will determine an assignment to the input variables—e.g., $\langle g, max, a\rangle = \langle 1, 2, -1\rangle$—violating the property, i.e., causing a division by 0.

The abilities to express properties as assertions, and to return an assignment to the input variables falsifying the property are the key points of the methodology presented in this paper.

3 Bounded Model Checking for Test Generation

Testing a (piece of) code means generating a set of tests that, when used as input to the program will either find a bug or make the programmer somehow more confident that the code does not present errors. A *test* is an assignment to the input variables of the program. So, for instance, given the program in Fig. 2, the set X of input variables is $\{g, max, a\}$ and a test t maps g,max and a to their respective domains, e.g., $t(g, max, a) = \langle 0, 1, 0\rangle$. The most used technique for automatic test generation is random testing [20]: for each input variable an admissible random value is generated and then the compiled code is run on the test. This operation is repeated until either an error is found or enough tests have been tried. Random Testing is automatic and simple to apply. However, it does not ensure an high coverage of the code, and it has been shown that it has quite low chances to determine faults revealed by a small percentage of the program input (called *semantically small faults* in [25]). Consider for example the following piece of code:

```
0    ...
1    if a == 0 then
2        block_1
3    else
4        block_2
5    ...
```

where it is assumed that a is an integer input variable not set in the lines executed before line 1.

The probability of exercising $block_1$ is in the order of $\frac{1}{n}$, where n is the maximum allowed value for integers, which depends on the particular architecture in use. So, for instance, if integer are represented with 64 bits, the probability of randomly generating a test exercising (and thus possibly finding a possible fault in) $block_1$ is $\frac{1}{2^{64}}$. In these cases, random testing is of little utility and, in practice, tests exercising

these blocks are manually generated by domain experts, with an implied associated cost.

On the other hand, tests covering these blocks can be automatically generated by CBMC by simply inserting an *assert(0)* at the beginning of the block to be covered, and running CBMC on the resulting code. In the case of the example in Fig. 2, putting an *assert(0)* between b_3 and s_4 and running CBMC on the instrumented code, an error trace is returned, corresponding to the test $t(g, max, a) = \langle 0, 1, 0 \rangle$.

4 Using Bounded Model Checking for Coverage Analysis of Safety-Critical Software in an Industrial Setting

The CENELEC EN50128 [15] is an European standard for the development of Railway applications. It concentrates on the methods which need to be used in order to provide software meeting 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 railway system requires SIL 4, meaning that the produced set of tests has to cover 100% of the code.

Consider a program C. In order to precisely define the notion of code coverage, we associate to C its control flow graph. The *control flow graph of C* is a graph representation of all paths that might be traversed by C during its execution. Each node in the graph represents a *basic block*, i.e., a piece of code without any jumps or jump targets; jump targets start a block and jumps end a block. Directed edges are used to represent jumps in the control flow. We also use two specially designated blocks: the *entry block*, through which control enters into the flow graph, and the *exit block*, through which all the control flow leaves. Figure 2 right shows the control flow graph associated to the function at the left. In the figure,

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** $a \neq 0$ **then**
s_3 $r = r + \frac{(g+2)}{a}$
b_3 **else**
s_4 $r = r + g + i$
s_5 $i{+}{+}$
s_6 $r = r * 2$
s_7 **return** r

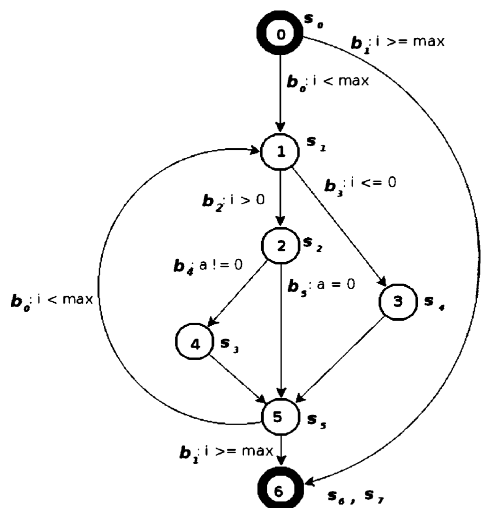

Fig. 2 An example of function (*left*) and its control flow graph (*right*)

1. Each node is labeled with a number and, for sake of clarity, is also annotated with the statements in the basic block it represents;
2. Each edge from b_s to b_t is possibly annotated with the condition that has to be satisfied in order to go from block b_s to block b_t;
3. Node 0 and node 6 represent the entry and exit block respectively.

In the following, we will not distinguish between a node in the control flow graph and the corresponding basic block in the program.

Consider a program C with input X and control flow graph G.

A *path* is a sequence $n_1, n_2, \ldots, n_m$ of nodes such that n_1 and n_m are the start and exit nodes of G respectively, and for all i with $1 \leq i < m$, $\langle n_i, n_{i+1} \rangle$ is an edge of G. 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*. The *set of statements* S_C *of* C is the set containing all the statements of C. In the same way, given the restriction on the syntax that we have, we can simply define the *set of decisions (or branches)* B_C *of* C as the set consisting of the conditions in the *if* statements of C and of their negation. For example, considering the program in Fig. 2,

$$S_C = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\}; \quad B_C = \{b_0, b_1, b_2, b_3, b_4, b_5\}.$$

For a test t it is possible to define S_t (resp. B_t) as the set of statements executed (resp. decisions satisfied) during the execution of the test t. For a set $T = \{t_1, ..., t_n\}$ of tests, we define the *set of statements (resp. decisions) executed by* T as:

$$S_T = \bigcup_{i=1}^{n} S_{t_i} \quad (\text{resp. } B_T = \bigcup_{i=1}^{n} B_{t_i})$$

For a set T of tests, the *percentage of statements covered by* T is:

$$sc_T = \frac{|S_T|}{|S_C|} \times 100,$$

while the *percentage of decisions covered by* T is:

$$bc_T = \frac{|B_T|}{|B_C|} \times 100.$$

For example, considering Fig. 2, given the set of tests $T = \{t_1, t_2\}$, with $t_1(g, max, a) = \langle 0, 1, 0 \rangle$ and $t_2(g, max, a) = \langle 0, 2, 0 \rangle$, t_1 and t_2 traverse the following set of statements and decisions:

$$\begin{array}{ll} S_{t_1} = \{s_0, s_1, s_4, s_5, s_6, s_7\}; & B_{t_1} = \{b_0, b_3, b_1\}; \\ S_{t_2} = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\}; & B_{t_2} = \{b_0, b_3, b_2, b_4, b_1\}; \end{array}$$

and then

$$S_T = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\}; \qquad B_T = \{b_0, b_3, b_2, b_4, b_1\}.$$

Thus,

$$sc_T = \frac{8}{8} * 100 = 100\%; \quad bc_T = \frac{5}{6} * 100 = 83.34\%.$$

In order to get the 100% of decision coverage required by the CENELEC EN50128, we have to extend the set T of tests, e.g. with $t_3(g, max, a) = \langle 0, 2, -1\rangle$ that covers the decision predicate b_5.

4.1 Test Generation via Path Predicate Construction in Ansaldo STS

As discussed in Section 3 random test generation often does not exercise each part of the code, i.e. it can not guarantee a 100% statement or decision coverage. In order to cover a specific path p, the standard approach followed in Ansaldo STS, is the generation of the *path predicate* associated to p, which is a system of conditions that the input variables should satisfy in order to traverse p. To understand how the path predicate is constructed, consider the path $p = \{0, 1, 3, 5, 6\}$ in the control flow graph of Fig. 2 right. This path traverses the decision predicates $\{b_0, b_3, b_1\}$: indeed the path predicate associated to p is not the conjunction of the decisions in the path, i.e.

$$((i < max) \wedge (i \leq 0) \wedge (i \geq max)). \tag{1}$$

In fact, (1):

- Is not a function of only input variables; and
- Does not have a solution, which would imply that p is unfeasible.

However, p can be traversed by $t(g, max, a) = \langle 0, 1, 0\rangle$. Indeed, while constructing the path predicate we ignored the execution of the statements in between the decisions, and this is the reason for the inconsistent result. Figure 3 shows the construction of the path predicate taking into account the statements in the path. Starting from the Fig. 3 left, for each line we report the line number (starting from 0), the statements and decisions traversed along the path, from the initial node to the end node. Then, starting from the bottom of the sequence, we delete each assignment $var = lhs$ and we substitute all the occurrences of var with lhs in the decisions below the deleted assignment, see, e.g., [16]. So, looking at Fig. 3 left, we start deleting line 7, without substitutions, and then line 5, substituting i in line 6 with $i + 1$: the result of these two operations is shown in Fig. 3 center, while the final result of the entire process is in Fig. 3 right, corresponding to the path predicate:

$$(0 < max) \wedge (0 \leq 0) \wedge (1 \geq max).$$

0	$i = 0$	0	$i = 0$	0	
1	$i < max$	1	$i < max$	1	$0 < max$
2	$g{+}{+}$	2	$g{+}{+}$	2	
3	$i \leq 0$	3	$i \leq 0$	3	$0 \leq 0$
4	$r = r + g + i$	4	$r = r + g + i$	4	
5	$i{+}{+}$	5		5	
6	$i \geq max$	6	$i + 1 \geq max$	6	$1 \geq max$
7	$r = r * 2$	7		7	

Fig. 3 Example of path predicate construction

The above formula is satisfied, e.g., by the test $t(g, max, a) = \langle 0, 1, 0\rangle$, which indeed exercises the path $p = \{0, 1, 3, 5, 6\}$.

In Ansaldo STS, the test generation process includes the construction of the path predicate and its solution. In more details, Ansaldo STS test generation process consists of two main steps:

- *Test generation*: a test is created by setting the input variables and it is added to the set of tests, initially empty. For the generation of the new test, a decision that has to be covered is individuated and a path predicate, associated to a path containing the decision, is manually constructed and solved. Solving the path predicate will give a test ensuring the execution of the decision. This new test is added to the set of tests.
- *Decision coverage computation*: the goal of the test-set generated is to cover the 100% of the decisions of the function under test. Ansaldo STS uses a tool called Cantata, see IPL: Cantata++, that calculates the percentage of decisions covered by a set of tests: if it returns 100%, then the testing generation phase ends. Otherwise, the entire process is repeated. Cantata, other then the percentage of decisions covered, also returns the decisions which are not covered by the test-set (if any), and these decisions are the ones considered in the previous step for the generation of new tests.

Notice that the construction of the path predicate and the computation of one solution is manually done in Ansaldo STS. Indeed, in the literature there are works showing that these two steps can be automatized at least to some extent (see for example [14, 19, 23, 30]) and that the difficult step is the solution of the derived equations, which may contain arbitrary operators and which may be non linear [9]. It is indeed an open question whether these approaches can be productively used in an industrial setting like Ansaldo STS.

It is however clear that the test generation process is very expensive for Ansaldo STS: Ansaldo STS has estimated that the manual generation of a single test requires *fifteen minutes* on average by a domain expert.

4.2 Test Generation via CBMC

In this section we show how the process in Section 4.1 can be fully automatized using CBMC. As we have already said in Section 3, assuming we want to generate a test covering a given block of the function under test, the basic and simple idea is to add an *assert(0)* in the block and then run CBMC. There are however a few subtleties that need to be addressed in order to make such simple idea productive. The methodology we used is presented in Fig. 4, which consists of three main steps:

1. *Code Instrumentation*: Consider a function f. In order to use CBMC for generating tests for f, CBMC requires (1) the existence of a function *main* invoking f, and (2) that at each function called by f is completely defined. For our goals, the *main* function, beside invoking f, has also to set the input values as to model possible user inputs. To this end, we use CBMC nondeterministic choice functions which have the prefix *nondet_*: for example, the function *notdet_int* (see Fig. 5 left) returns an arbitrary integer. The same functions are used as *stubs* for the functions called by f and of which we do not have a definition. Consider now the task to generate a test set for f covering all the decisions of f. The simple idea

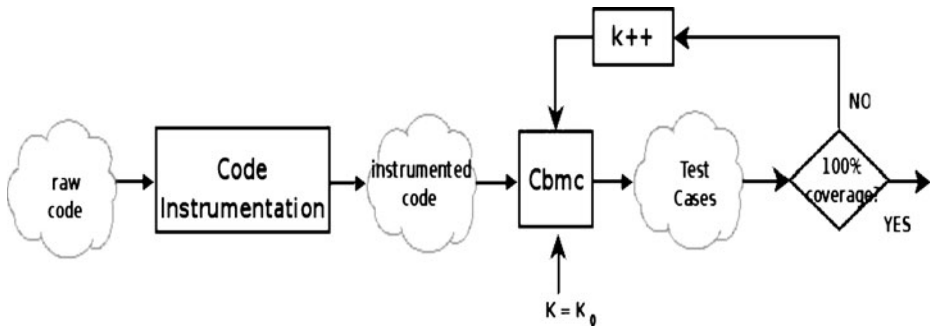

Fig. 4 Testing process with CBMC

to insert all the *assert(0)* together, one per block, and then run CBMC does not work: during the generation phase CBMC will stop at the first assertion which gets violated. An alternative is to insert an *assert(0)* at a time generating one file for each assertion introduced. However, in this way many files will be generated, one for each test, causing difficulties in the management of the test cases. A better solution is to have an *assert(0)* inserted in each block and condition the executability of each *assert(0)* to the value of an input parameter of CBMC. For

```
int MAIN(int argc, char * argv[ ])
    max = NONDET_INT()
    g = NONDET_INT()
    int a = NONDET_INT()
    return FUT(a)
```

```
int FUT(int a)
           ASSERT_1
s0         int r = i = 0
b0, b1     while i < max do
           ASSERT_2
s1            g++
b2            if i > 0 then
           ASSERT_3
s2               a++
b4               if a ≠ 0 then
           ASSERT_4
s3                  r = r + (g+2)/a
b5               else
           ASSERT_5
b3            else
           ASSERT_6
s4               r = r + g + i
s5            i++
           ASSERT_7
s6         r = r * 2
s7         return r
```

Fig. 5 An example of instrumented function

this task we use CBMC ability to handle conditionals, i.e., *#ifdef*s. Thus, if n is the number of decisions, we introduce n macros like:

#ifdef ASSERT_i
assert(0)
#endif

with $i = 1, \ldots, n$. Figure 5 presents the instrumented version of the code in Fig. 2. In Fig. 5 right, we write *ASSERT_i* as an abbreviation of the corresponding conditional macro.

2. *Test Generation*. Given the instrumented code, CBMC is run n times, one for each $i = 1, \ldots, n$, with the following command line

   *C*BMC -D ASSERT_*i file*.c–unwind *k*–no-unwinding-assertions

   where

   (a) *D ASSERT_i* causes the inclusion of an *assert(0)* in the *i*-th branch,
   (b) *File.c* is the name of the file with the function to test,
   (c) *Unwind k* fixes the unwinding bound to *k*: initially *k* is fixed to 1 and is incremented after each iteration (see Fig. 4), and
   (d) *No-unwinding-assertions* option prevents CBMC from including assertions checking the non existence of inputs causing the execution of loops for more than *k* times: CBMC, in the presence of multiple assertions, will try to violate an arbitrary one, and we want CBMC to focus on violating the assertion we added, corresponding to a test case for the *i*-th branch.

   From each run returning an assertion violation, a counter-example is produced from which a test is extracted.

3. *Coverage Analysis*. It can be the case that a run of CBMC fails to generate a test in the previous step. In this case, if CBMC did not exit abnormally (e.g., because of memory out of the SAT solver), this implies that the corresponding decision cannot be covered with the given bound k. For example, considering the program in Fig. 2 with k set to 1, then the statements s_2, s_3 and s_4 can not be covered. To check if we get the desired 100% decision coverage with the test set T so far generated, a coverage analysis process with Cantata is performed: if the percentage of decisions covered by T is less than 100% then k is incremented and the testing generation phase can be executed again; otherwise the process stops.

Each block is completely automatic. Of course, the entire process has to be monitored, e.g., in order not to loop forever in the case of a program with an unfeasible block.

4.3 Experimenting with the ERTMS

Nowadays trains are equipped with up to six different extremely costly navigational systems. A train crossing from one European country to another must switch the operating standards as it crosses the border. The ERTMS [18] 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, 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 the radio system, based on the standard GSM, used to exchange voice and data information between the track and the train.

Ansaldo STS, as a partner 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 100% of the decisions. 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 five interconnected modules of the EVC and we applied the automatic test generation methodology discussed in the previous section.

Figure 6 left, presents the five modules under test and their interconnections, while the figure on the right presents a call graph of one of the functions under test. In the call graph there is an arrow between the generic functions f_i and f_j if in f_i there is a function call to f_j. The function without any input connections, like the function in the first box on the left, represents the function under test and the functions in a grey box are defined in the same module of the function under test, whereas functions in a white box are defined in other modules. The five modules under test contain 74 different functions presenting no recursive calls, and the maximum number of possible iterations of each loop is known a priori. For industrial copyright purposes, we omit each module's name, substituting them with m_i, $i = 1, \ldots, 5$. The five modules contain more than 10,000 lines of code, while the entire EVC project contains more than 100,000 lines of code. Table 1 shows the results of the automatic test generation on the five modules of the EVC. In the table,

1. Column "Mod" reports the name of the module;
2. Column "#f" shows the number of functions of each module;
3. In column "CBMC" we report the number of tests (subcolumn "#t") generated by CBMC according to the methodology described in the previous section, and the time (all the times are in minutes) needed to generate them (subcolumn "#m");
4. Column "Ansaldo STS" shows the number of tests manually generated by domain experts (subcolumn "#t") and an estimation of the time needed to generate them (subcolumn "#m");
5. Column "Parasoft 50" (resp. "Parasoft 100", resp. "Parasoft 300") shows the results for Parasoft C++ Test 6.5 (Parasoft in short from now on) when set to automatically generate 50 (resp. 100, resp. 300) tests per function. As before, we show the number of generated tests (subcolumn "#t"), an estimation of the time needed to generate them (subcolumn "#m"), and the percentage of the decisions covered by the test set (subcolumn "#C").

Before going to the results, a few remarks are in order. First, for CBMC and Ansaldo STS we do not report the decision coverage obtained because all the test

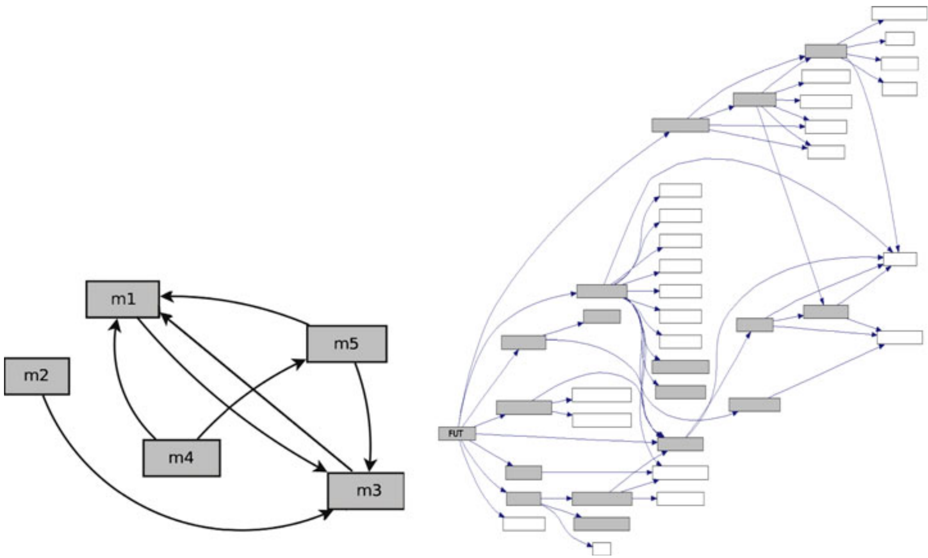

Fig. 6 Connection between the five modules under test of the ERTMS (*left*); a call graph of a function of one of the modules (*right*)

sets reach the target 100%. Second, as we said, the tests are automatically generated in the case of CBMC and Parasoft, and manually generated in the case of Ansaldo STS: thus, the subcolumn "#m" simply answers to the question "how much time is needed to generated the tests" and does not take into account the diversity of resources used by the two (automatic vs manual) approaches. Third, for Ansaldo STS and Parasoft, we said that the reported timings are estimates. For Ansaldo STS, we recall that Ansaldo STS estimates that each manually generated test requires an average of 15 min: thus, the result is simply obtained by multiplying the number of tests per 15. In the case of Parasoft, we are in the presence of an integrated tool running on Windows for which it is difficult to get accurate timings. Finally, given that Parasoft may select some values for input variables at random, we run Parasoft five times on each module, and we show the results for the run giving the highest decision coverage.

Considering the data in the table, we can see that for each module the number of tests automatically generated by CBMC is close to the double of the number of tests

Table 1 Experimental analysis on the five modules of the EVC

Mod	#f	CBMC		Ansaldo STS		Parasoft 50			Parasoft 100			Parasoft 300		
		#t	#m	#t	#m	#t	%C	#m	#t	%C	#m	#t	%C	#m
m_1	19	148	20	64	960	806	82	6	1,606	81	10	4,806	81	22
m_2	8	47	24	26	390	400	87	5	800	85	7	2,400	87	18
m_3	13	193	104	80	1,200	516	81	6	1,016	83	10	3,016	87	22
m_4	18	184	22	110	1,650	900	66	4	1,800	70	6	5,304	76	18
m_5	16	185	28	105	1,575	800	69	6	1,600	74	9	4,800	83	18
Total	74	757	198	385	5,775	3,422		27	6,822		42	20,326		98

manually generated by Ansaldo STS. On the other hand, the time spent to manually generate the tests is more than an order of magnitude higher than the time required by CBMC: it is also clear that the resources involved in the process (domain experts for manual generation and computers for the automatic generation) have different costs. Looking at the totals in the last line, we can see that CBMC will generate 757 tests in less than 4 h, while the test manually generated are only 385, but the time spent is almost 100 h.

The opposite holds if we compare CBMC with Parasoft 50|100|300: CBMC generates less tests but in more time than Parasoft. Here however the main result is that Parasoft never reaches the target 100% coverage and, looking at the results, it seems likely that even (reasonably) increasing the number of tests which are generated per function, full decision coverage won't be reached, at least for the first two modules. About Parasoft, notice the apparently contradictory result that for the first module we get an higher coverage when generating 50 tests per function, than when considering 100 or even 300 tests per function: this is not surprising if we take into account that in Parasoft generation process, there is a certain degree of randomness and it is possible to get (by chance, as—we suspect—in this case) a higher coverage by generating less tests. Also notice that the number of tests reported—e.g., for Parasoft 50—is not always equal to the number of functions times 50. Indeed, for some functions, Parasoft may generate less tests than 50 and obtain full coverage. This is the case, e.g., for three functions in the first module which take in input a pointer and whose body contains only two decisions, corresponding to a test checking if the input pointer is null or not: for these functions, Parasoft generates just two tests, one being the null pointer.

Summing up, given the target to get full decision coverage, the methodology we proposed based on CBMC appears to be far more productive than the manual generation by domain experts. Parasoft and, more in general, automatic test generation methods using random testing, fail to get full decision coverage on our benchmarks and thus cannot productively be used, even when coupled with manual generation by domain experts. To substantiate this claim, consider Table 2 showing the results of the different systems on nine functions of the EVC.

In Table 2, (sub-)columns' names have the same meaning as in Table 1: the two newly introduced labels "Fun" and "#s" stand for the function name (each of the form f_{j,m_i}, where m_i is the name of the module where the function is defined) and for the number of seconds taken to generate the tests, respectively. As it can be observed, Parasoft always fails to cover all the decisions of each function, and the

Table 2 Experimental analysis on some functions of the EVC

Fun	CBMC		Ansaldo STS		Parasoft 300
	#t	#s	#t	#s	%C
f_{1,m_1}	27	202	7	6,300	92%
f_{1,m_3}	35	576	14	12,600	86%
f_{2,m_3}	38	367	15	13,500	84%
f_{1,m_4}	12	66	9	8,100	90%
f_{2,m_4}	29	223	18	16,200	89%
f_{3,m_4}	21	125	12	10,800	89%
f_{4,m_4}	29	271	14	12,600	67%
f_{1,m_5}	17	161	10	9,000	75%

manual generation of a single test (estimated in 900 s as usual) is going to cost more than the time taken by CBMC to generate all the tests. It worths remarking that, as before, for each function we run Parasoft five times and show the results for the run obtaining the highest decision coverage.

It can be argued that the tests generated by CBMC are just *skeletons*, i.e., some values for the input variables. The *real* tests, i.e. the ones containing the drivers and stubs for the concrete execution, are not provided by CBMC. Ansaldo STS has developed a completely automatic process to obtain the real tests from the skeletons, and the time needed by this process is insignificant with respect to the time spent for the generation.

Summing up, these results clearly indicate that the methodology proposed based on CBMC leads to a dramatic increase in the productivity of the entire Software Development process by substantially reducing the efforts associated to the testing phase.

5 Conclusions and Related Work

In this paper we have shown how BMC can be successfully applied to the Software Verification process in industry. In particular, we have shown how CBMC can be successfully used for the automatic generation of a set of tests covering the 100% of decisions as required by the CENELEC EN50128 guidelines. Our experiments report that the use of CBMC led to a dramatic increase in the productivity of the entire Software Development process, by substantially reducing the time spent, and consequently, the costs of the testing phase. To the best of our knowledge, this is the first time that BMC techniques have been used in an industrial setting for automatically generating tests achieving full coverage of Safety-Critical Software. The positive results demonstrate the maturity of Bounded Model Checking techniques for automatic test generation in industry.

The use of BMC for automatic test generation is not new in the field of verification of circuits and microprocessor design [6, 31]. Nevertheless, as far as we know, this is the first time that BMC has been applied in the field of coverage analysis of safety critical software.

Pex [12, 28] is a tool for Automatic Test generation, developed at Microsoft Research, which helps developers to write PUTs (Parameterized Unit Tests) [29] in .NET language. For each PUT, Pex uses dynamic test-generation techniques to compute a set of input values that exercises all the statements and assertions in the program. The main difference between Pex and our approach is that Pex is not completely automatic (PUTs are written by hand) and does not guarantee the 100% of decision coverage. As a side remark, Pex is designed for the .NET framework, and uses a constraint solver as main engine, called Z3 [13].

Another approach that uses symbolic execution is *DART* (Directed Automated Random Testing) [20]. DART is a tool to automatically test the software that combines (1) automatic interface extraction, (2) random test generation, and (3) dynamic analysis during the execution of the random test, in order to systematically drive the generation of the new tests along alternative program paths. It differs from our approach since (1) it works on programs that compile; (2) it does not guarantee 100% coverage, and (3) it uses constraint solving techniques to find alternative paths.

Symbolic execution is also used in Concolic Testing [27], where symbolic execution is combined together with the concrete execution to automatically generate test inputs. In particular, during the execution of an either randomly or manually generated test, all the decision constraints resolved are collected to form a path constraint. Then, one of the decision constraints is negated to form a new path constraint that is passed to a constraint solver (symbolic execution). If the constraint solver returns a new assignment to the input variables, a new path constraint is generated, otherwise if the path constraint can not be solved, one or more decisions are substituted by their concrete values computed during the concrete execution. The method differs from our approach in many ways, e.g., it makes an essential use of concrete executions. Further, it does not guarantee the target 100% coverage.

Another approach, similar to Concolic Testing and DART/Pex is presented in a tool called KLEE [5]. KLEE explores a path along the control flow graph of the program, collecting its path constraint. Then the path constraint is passed to a constraint solver that returns an assignment to the input variables, if any, which makes true the path constraint. KLEE will continue until all the statements of the program are covered by the set of tests computed. KLEE has been used to determine high coverage tests for several complex programs, but it has not been used in a productive industrial environment. Further, KLEE aims to statement coverage, while our goal is decision coverage. Finally, KLEE, like previous already mentioned approaches, use constraint solving techniques, while our approach is based on bounded model checking and SAT.

A more similar approach using Bounded Model Checking as symbolic executor is presented in TestEra [22], a tool for automatic test generation for Java Programs. Given a method in Java (source code or bytecode), a formal specification of the pre- and post-conditions of the method and a bound limit on the size of the test cases to be generated, TestEra automatically generates all nonisomorphic test inputs up to the given bound. Specifications are first order logic formulae, and as enabling technology, TestEra uses the Alloy [21] tool set, which provides an automatic SAT-based tool for analyzing first-order logic formulae. TestEra is not fully automatic requiring a very skilled tester for writing pre- and post-conditions of each function (first-order logic formulae) and it does not guarantee the target 100% decision coverage.

There are also a few works using Bounded Model Checking for automatic generator tests for software analysis [1, 4]. However, these approaches are not completely automatic and have a different focus: in [4] and [1] the goal of the generated tests is to satisfy a given property of the program under test and not to obtain a 100% decision coverage.

References

1. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating Tests from Counterexamples. In: ICSE, pp. 326–335. IEEE Computer Society (2004)
2. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 118–149 (2003)
3. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, R. (ed.) TACAS. Lecture Notes in Computer Science, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)

4. Black, P.E., Ammann, P., Ding, W., N. I. of Standards, T. (U.S.): Model checkers in software testing. U.S. Dept. of Commerce, Technology Administration, National Institute of Standards and Technology, Gaithersburg (2002)
5. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) OSDI, pp. 209–224. USENIX Association (2008)
6. Chockler, H., Kupferman, O., Kurshan, R.P., Vardi, M.Y.: A practical approach to coverage in model checking. In: Berry, G., Comon, H., Finkel, A. (eds.) Computer aided verification. 13th international conference, CAV 2001, Paris, France, 18–22 July 2001, Proceedings. Lecture Notes in Computer Science, vol. 2102, pp. 66–78. Springer, Heidelberg (2001)
7. Clarke, E.M., Kroening, D., Ouaknine, J., Strichman, O.: Completeness and Complexity of Bounded Model Checking. In: Steffen, B., Levi, G. (eds.) VMCAI. Lecture Notes in Computer Science, vol. 2937, pp. 85–96. Springer (2004)
8. Clarke, E.M., Kroening, D., Yorav, K.: Behavioral consistency of C and verilog programs using bounded model checking. In: DAC, pp. 368–371. ACM (2003)
9. Cooper, D.: Theorem proving in arithmetic without multiplication. In: Meltzer, B., Michie, D. (eds) Machine Intelligence, vol. 7. Edinburgh University Press, Edinburgh (1972)
10. Copty, F., Fix, L., Fraer, R., Giunchiglia, E., Kamhi, G., Tacchella, A., Vardi, M.Y.: Benefits of bounded model checking at an industrial setting. In: Berry, G., Comon, H., Finkel, A. (eds.) Computer aided verification. 13th international conference, CAV 2001, Paris, France, 18–22 July 2001, Proceedings. Lecture Notes in Computer Science, vol. 2102, pp. 436–453. Springer, Heidelberg (2001)
11. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4), 451–490 (1991)
12. de Halleux, J., Tillmann, N.: Parameterized unit testing with Pex. In: Beckert, B., Hähnle, R. (eds.) Tests and proofs, second international conference, TAP 2008, Prato, Italy, 9–11 April 2008. Proceedings. In: Lecture Notes in Computer Science, vol. 4966, pp. 171–181. Springer, Heidelberg (2008)
13. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R. Rehof, J. (eds.) TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
14. Deason, W.H., Brown, D.B., Chang, K.-H., Cross, J.H.: A rule-based software test data generator. IEEE Trans. Knowl. Data Eng. **3**(1), 108–117 (1991)
15. EC: European committee for electrotechnical standardization. In: Railway Applications—Communication, Signalling and Processing Systems - Software for Railway Control and Protection Systems (2008)
16. Edvardsson, J.: A survey on automatic test data generation. In: Proceedings of the Second Conference on Computer Science and Engineering in Linkö. pp. 21–28 (1999)
17. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT. Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer, Heidelberg (2003)
18. ERTMS: The official Website: http://www.ertms.com/
19. Ferguson, R., Korel, B.: The chaining approach for software test data generation. ACM Trans. Softw. Eng. Methodol. **5**(1), 63–86 (1996)
20. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Sarkar, V., Hall, M.W. (eds.) PLDI, pp. 213–223. ACM (2005)
21. Jackson, D., Shlyakhter, I., Sridharan, M.: A micromodularity mechanism. In: ESEC/SIGSOFT FSE, pp. 62–73 (2001)
22. Khurshid, S., Marinov, D.: TestEra: specification-based testing of java programs using SAT. Autom. Softw. Eng. **11**(4), 403–434 (2004)
23. Meudec, C.: ATGen: automatic test data generation using constraint logic programming and symbolic execution. Softw. Test., Verif. Reliab. **11**(2), 81–96 (2001)
24. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: DAC, pp. 530–535. ACM (2001)
25. Offutt, A.J., Hayes, J.H.: A semantic model of program faults. In: ISSTA, pp. 195–200 (1996)
26. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. J. Symb. Comput. **2**(3), 293–304 (1986)
27. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 263–272 (2005)

28. Tillmann, N., de Halleux, J.: Pex-White box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) Tests and proofs, second international conference, TAP 2008, Prato, Italy, 9–11 April 2008. Proceedings. In: Lecture Notes in Computer Science, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
29. Tillmann, N., Schulte, W.: Parameterized unit tests. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 253–262 (2005)
30. Tracey, N., Clark, J.A., Mander, K.: Automated program flaw finding using simulated annealing. In: ISSTA, pp. 73–81 (1998)
31. Vedula, V.M., Abraham, J.A., Ambler, T.P., Aziz, A., Chase, C.M., Tupuri, R.S., Vedula, M.A., Tech, B.: HDL slicing for verification and test (2003)