Exploiting Safety Properties in Bounded Model Checking for Test Cases Generation of C Programs

Herbert Rocha, Lucas Cordeiro, Raimundo Barreto and José Netto

[1]Universidade Federal do Amazonas
Av. Rodrigo Octávio, 3000, Coroado, Manaus/Amazonas

{herberthb12, lucasccordeiro}@gmail.com,{rbarreto, jnetto}@dcc.ufam.edu.br

Abstract. *The use of computer-based systems in several domains has increased significantly over the last years so that software verification now plays an important role in ensuring the overall product quality. The value of the counterexample and safety properties generated by Bounded Model Checkers to create test case and to debug these systems is highly recognized. In this paper, we describe a method to integrate the bounded model checker ESBMC with the CUnit framework. This method aims to extract the safety properties generated by ESBMC to generate automatically testcases using the rich set of assertions provided by the CUnit framework. We show the effectiveness of our proposed method over publicly available benchmarks.*

Resumo. *O uso de sistemas computacionais em diversos domínios têm crescido significativamente nos últimos anos, desta forma a verificação de software desempenha um papel importante para garantir a qualidade geral do produto. O valor dos contra exemplos e propriedades de segurança gerados pelos Bounded Model Checkers para a geração dos casos de teste e depuração destes sistemas são altamente reconhecidos. Neste artigo, descrevemos um método para integrar o ESBMC com o framework CUnit. Este método visa extrair as propriedades de segurança geradas pelo ESBMC e gerar automaticamente casos de testes usando um rico conjunto de assertivas providas pelo CUnit. Demonstramos a eficácia do método, aplicando-o sobre benchmarks públicos.*

1. Introduction

Nowadays, software applications need to be developed quickly and meet a high-level of quality. Formal verification plays an important role to ensure predictability and dependability in the design of critical applications. Consequently, the application of verification and testing are indispensable techniques to the development of high-quality software. However, there is usually a high cost involved in the preparation, execution and management of tests. One way to deal with such problems is to integrate formal verification techniques with test environments. Model checking has been an interesting area of research in formal verification for many years. In the last decades, the use of models has been adopted in different software categories, promoted by software technologies such as UML [Borges and Mota 2007] and Model-Driven Testing [Petrenko and Alexandre 2001]. This scenario has also motivated software engineers to apply model checking [Beyer et al. 2007]. Model checking is defined as a strategy for verifying the systems' correctness by checking whether the system meets desired properties based on its representation in a formal model [Baier and Katoen 2008].

The main challenge in model checking is how to deal with both the state space explosion problem and the lacking of integration with other test environments more familiar to designers. In order to tackle these problems, one possible solution is to explore features already provided by the model checking (for instance, identification of safety properties) for test case generation. According to [Baier and Katoen 2008] safety properties are often characterized as "*nothing bad should happen*". As example, the mutual exclusion property is a typical safety property. In this way, the violation of a safety property can be detected by monitoring the runtime system execution. Such monitoring is usually done in a controlled manner, taking into account both system inputs, and several execution paths.

In this paper, we describe a method to integrate the model checker ESBMC (Efficient SMT-Based Bounded Model Checker)[Cordeiro et al. 2009b] with the CUnit framework[1]. The integration of these two environments aims to ensure software quality by exploiting formal verification and tests without obtaining a memory explosion. In particular, this method extracts the safety properties generated by the ESBMC to generate automatically test cases using the rich set of assertions provided by the CUnit. We show the effectiveness of our proposed method over publicly available benchmarks. We advocate that exploiting the integration between a testing framework and a model checker allows us to go deeper into the program verification.

2. Background

This section presents the ESBMC and discuss about safety properties and software testing technique.

2.1. Model Checking with ESBMC

Model Checking is one of the most used formal method for the verification of systems, which generates an exhaustive search in the state space model to determine whether a given "property" is valid or not [Baier and Katoen 2008]. There is a special kind of model checking algorithm called *bounded model checking* (BMC) based on Boolean Satisfiability (SAT), which has been introduced as a complementary technique to Binary Decision Diagrams for alleviating the state explosion problem. The basic idea of BMC is to check (the negation of) a given property at a given depth: given a transition system *M*, a property ϕ, and a bound *k*, BMC unrolls the system *k* times and translates it into a verification condition (VC) ψ such that ψ is satisfiable if and only if ϕ has a counterexample of depth less than or equal to *k*. Standard SAT checkers can be used to check if ψ is satisfiable. In order to cope with increasing system complexity, SMT (Satisfiability Modulo Theories) solvers can be used as back-ends for solving the VCs generated from the BMC instances. ESBMC[2] is a bounded model checker for embedded ANSI-C software based on SMT solvers, which allows: (i) to verify single- and multi-threaded software (with shared variables and locks); (ii) to reason about arithmetic under- and overflow, pointer safety, array bounds, atomicity and order violations, deadlock, data race, and user-specified assertions; (iii) to verify programs that make use of bit-level, pointers, structs, unions and fixed-point arithmetic. ESBMC uses a modified version of the CBMC[3] front-end to parse the ANSI-C code and to generate the VCs via symbolic execution. ESBMC provides three approaches

[1] Available at: http://cunit.sourceforge.net/

[2] Available at http://users.ecs.soton.ac.uk/lcc08r/esbmc/

[3] Available at http://www.cprover.org/cbmc/

lazy, schedule recording, and underapproximation and widening (UW) to model check multi-threaded software. This paper aims to exploit features provided by ESBMC in order to propose an alternative approach to test-based design, since BMC alone has limitations, such as the state space explosion to check the satisfiability of the VCs by using off-the-shelf SMT-solvers.

2.2. Exploiting Safety Properties in Software Testing Strategies

Software testing is the process of executing a program with the goal of finding faults [Myers and Sandler 2004]. A successful test is the one that can determine the test cases for which the program under test fails. A test case consists of an analysis of test data associated with an expected result of processing according to the software specification. Unit tests are typically written based on a set of test cases to ensure that the code meets its design and behaves as expected. There are lots of tools for developing unit tests. Such tools allow software engineers to create unit tests in a more efficient and simplified way by favouring better organization and code reuse. CUnit is an unit testing framework for C programs that provides a set of assertions for testing logical conditions. The success or failure of these assertions is tracked by the framework, and can be viewed when a test run is complete. Each assertion tests a single logical condition, and fails if the condition evaluates to *FALSE*. The problem that we address in this paper is *how to create test cases aimed at checking safety properties?* We thus propose an approach to create test cases guided by the safety properties that are generated automatically by ESBMC. ESBMC generates verification conditions for the following set of safety properties: (1) VCs to check for arithmetic overflow and underflow; (2) VCs to check for out-of-bounds array indexing; (3) VCs to check for pointers safety; and (4) VCs to check for dynamic memory allocation. The proposed method aims to transform each property identified by ESBMC in an assertive in CUnit in order to obtain, as a final result, a new instance of the analyzed code, containing test cases directly linked to the safety properties of the code.

3. Related Work

There are several tools and methods for generating test cases. GATeL [Marre and Arnould 2000] requires the SUT (System Under Test) or a complete specification of the SUT, an environment description, and a test objective (safety or liveness property) in order to generate the test cases. AsmL [Barnett et al. 2003], TestComposer, and AutoLink [Schmitt et al. 2000] are strongly linked to a specific domain or language. In [Cadar and Engler 2005] is presented a technique that uses code to automatically generate its own test cases at run-time by using a combination of symbolic and concrete (i.e., regular) execution. The first contribution is the observation that code can be used to automatically generate its own potentially highly complex test cases. Instead of running the code on manually-constructed concrete input, this technique runs it on symbolic input. These observations say what legal values (or ranges of values) the input could be, and thus generate test cases by solving these constraints for concrete values. These tests are called *execution generated testing* (EGT). However, this work does not take into account BMC and how to solve the memory and solver limitations.

4. Proposed Method

This section describes the main steps of our proposed method that aims to explore safety properties generated by ESBMC in order to generate test cases of C programs. The pro-

posed method consists in the following steps: *(i)* Identification of safety properties; *(ii)* Safety Properties Information Collection; *(iii)* Asserts Inclusion; *(iv)* Implementing Unit Test in CUnit Framework; and *(v)* Running CUnit tool. Figure 1 shows an overview of our proposed method. It is worth saying that all such steps are automatically performed by specific tools. In order to explain the main steps of the proposed method we adopt the code shown in Figure 2.

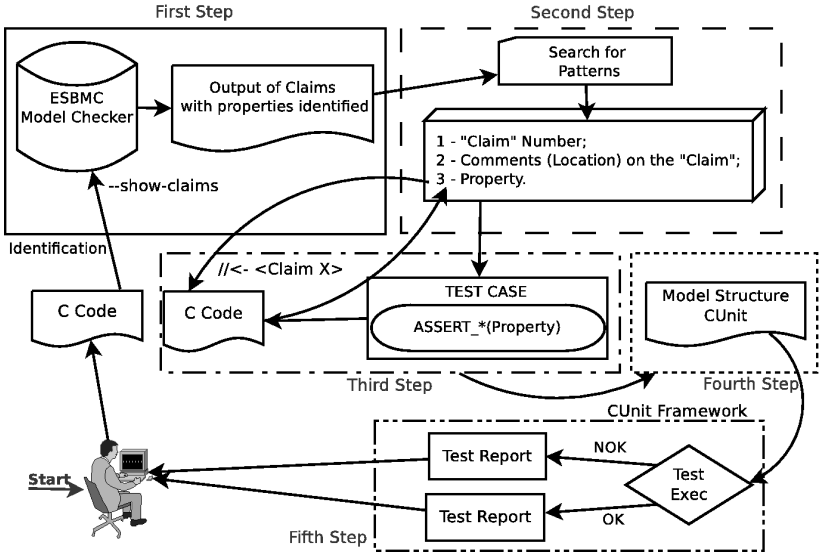


Figure 1. Flow structure of the proposed method.

```
1  #include <stdio.h>
2  int a[5], b[6];
3  int main(){
4   int i, j, temp; a[0]=1;
5   for(i=1; i<5; i++) {
6     a[i]= a[i-1] + i; b[0] = 1;
7     temp = a[i]*(i+1);
8     for(j=1;j<temp;j++) { b[j] = b[i-1]+(temp*2); }
9   }
10 }
```

Figure 2. C code for describing the proposed method steps.

4.1. *First step*: Identification of Safety Properties

We adopt ESBMC for identification of safety properties that receives a C program as input parameter and option `--show-claims`, which shows all found safety properties that ESBMC think it can be violated. In the bounded model checking context, a `claim` is the same as a property. Examples of safety claims are: buffer overflow, division by zero, arithmetic overflow, and so on. Therefore, a claim may be violated in some execution path. It is worth noting that automatically generated claims not necessarily correspond to bugs, but they are just potential flaws. Whether one of these claims corresponds to a bug needs to be determined by further analysis. Each safety property (or claim) generated by ESBMC contains the following information: property identification, location,

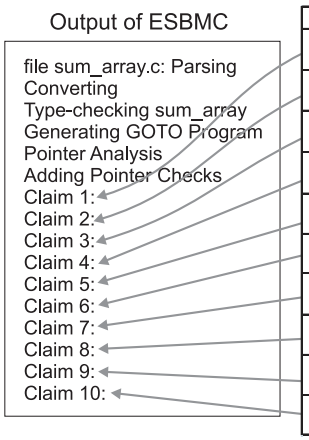
and the `assert` to be checked. Figure 3 shows an example of a claim automatically generated. In this example, claim 1 states a potential lower bound of array "a" in file *sum_array.c* at line 9 of the function *main*. All claims are stored in a special text file called "`result_ESBMC.txt`" so that important information can be extracted in the next step.

```
herbert@nbh /media/$ esbmc sum_array.c --show-claims
file sum_array.c: Parsing
Converting
Type-checking sum_array
Generating GOTO Program
Pointer Analysis
Adding Pointer Checks
Claim 1:
  file sum_array.c line 9 function main
  array `a' lower bound
  i >= 0
```

Figure 3. Proposed method first step.

4.2. *Second step*: Safety Properties Information Collection

The second step analyzes the text file produced in the step 1 to collect several important information needed in the next steps, such as: *(i)* claims, *(ii)* comments on the claim, *(iii)* the line of code where the claim occurred, and *(iv)* the property identified by that claim. The text file "`result_ESBMC.txt`" is given as input of this step where there may be several claims but the file organization follows the same pattern. We adopt regular expression to find all information. As result of this step is produced a new text file called "`result_claims.txt`" that stores several information as depicted in Figure 4.



Claims	Comments	Line in the Code	Property
Claim 1	file sum_array.c line 9 function main array `a' lower bound	9	i >= 0
Claim 2	file sum_array.c line 9 function main array `a' upper bound	9	i < 5
Claim 3	file sum_array.c line 9 function main array `a' lower bound	9	-1 + i >= 0
Claim 4	file sum_array.c line 9 function main array `a' upper bound	9	-1 + i < 5
Claim 5	file sum_array.c line 11 function main	11	i >= 0
Claim 6	file sum_array.c line 11 function main	11	i < 5
Claim 7	file sum_array.c line 13 function main	13	j >= 0
Claim 8	file sum_array.c line 13 function main	13	j < 6
Claim 9	file sum_array.c line 13 function main	13	-1 + i >= 0
Claim 10	file sum_array.c line 13 function main	13	-1 + i < 6

Figure 4. Applying the second step.

4.3. *Third step*: Asserts Inclusion

The information collected in the second step will serve as a basis for creating test cases in the C source code. This step includes assertions in the code with the respective safety property generated by ESBMC. This step is divided in two phases: (i) to include each claim in the C source code as comments; (ii) Add an assertive containing the (maybe violated) safety property to each claim comment added in phase (i). The first phase uses the file "`result_claims.txt`" to get the line where the claim has ocurred, and enter

in the C source code a comment about all claims. Therefore, this step adds at the end of the line a comment "`//<-`" and, for each claim, "`<Claim X>`", where X is the claim number. Figure 5 shows the results of claims identification. The second phase consists in finding each claim previously added as a comment, and adding an assertive containing their corresponding safety property in a line before the line code identified with the respective *claim*. Each line of the C code that has this kind of comment is analyzed in order to find what is the claim number. This claim number is checked in the file "`result_claims.txt`" to obtain their respective safety property, and other relevant information, to be included in the C source code as an assertion. Figure 6 shows an example based on the code of Figure 2.

```
1 for( i=1; i<5; i++){
2  a[i]= a[i-1] + i;//<- <Claim 1> <Claim 2> <Claim 3> <Claim 4>
3  b[0] = 1;
4  temp = a[i]*(i+1);//<- <Claim 5> <Claim 6>
5  for(j=1;j<temp;j++){
6   b[j]=b[i-1]+(temp*2);//<- <Claim 7> <Claim 8> <Claim 9>
7  }
8 }
```

Figure 5. C code with added comments in the third step.

```
1 //Claim 1: file sum_array.c line 6 //array 'a' lower bound
2 CU_ASSERT(i >= 0);
3 //Claim 2: file sum_array.c line 6 //array 'a' upper bound
4 CU_ASSERT(i < 5);
5 a[i]= a[i-1] + i ; b[0] = 1 ;//<- <Claim 1> <Claim 2>
```

Figure 6. C code with assertions of third step.

4.4. *Fourth step*: Implementing Unit Test in CUnit Framework

This step changes the C code output file of the third step to a new file suitable to be applied in the CUnit framework. This new file is built by a template and contains: (i) include files specific for CUnit and C source code; (ii) CUnit configuration functions; (iii) functions to be tested; and (iv) new main function that runs CUnit. Figure 7 shows the template phases and the respective test code for running unit tests using CUnit. The CUnit include files are directly obtained by the template file. The C source code include files are directly copied from the C source code. The CUnit configuration functions are directly obtained by the template file. The functions to be tested are filled by the main function of the C source code changing the function name from "main" to "testClaims". The new main function area is directly obtained by the template file. The result is a new file that is ready to be tested by CUnit framework.

4.5. *Fifth step*: Running CUnit Tool

In the last step, the new C code is submitted for execution with the CUnit. In this case, CUnit performs tests on the code that contains the test cases that was generated during the

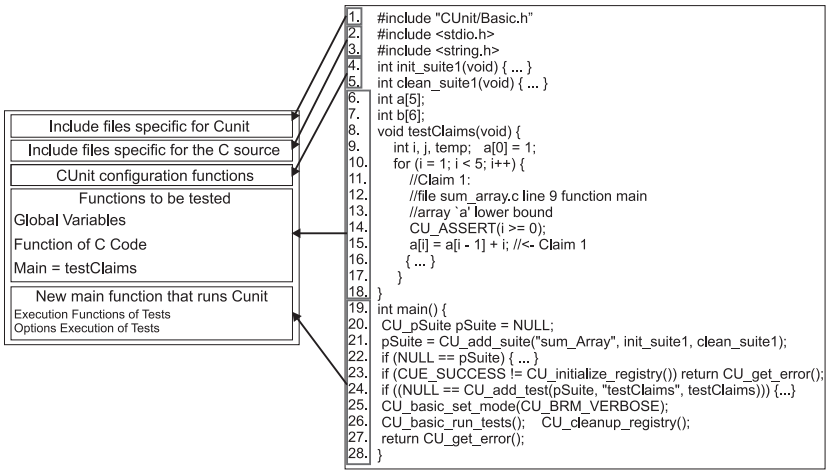


Figure 7. Test code for running unit tests using Cunit framework.

method application, by abstraction of safety properties. Initially, test cases are analyzed and each one can be approved or failed. Each failed test is reported by the testing framework. Figure 8 shows the result of applying the proposed method in the C code shown in Figure 2. As we can see, there is 1 test with 404 asserts where 77 have failed.

```
Suite: sum_Array
  Test: testClaims ... FAILED
    sum_Array.c:86  - j < 6
--Run Summary: Type      Total      Ran  Passed  Failed
               suites        1        1     n/a       0
               tests         1        1       0       1
               asserts     404      404     327      77
```

Figure 8. Output of Cunit.

5. Experimental Results

This section shows and analyzes results obtained with the proposed method when applied to the verification of standard ANSI-C benchmarks. Our experiments were conducted on an Intel Core 2 Duo CPU, 2Ghz, 3GB RAM with Linux OS. The steps of the proposed method have been implemented using the ESBMC v1.11, and the framework CUnit v2.1. Table 1 details the ANSI-C benchmarks used in our experiments. The first column contains the application number; the second columns describes the module name composed by the respective benchmark, C program, and the input (e.g., deterministic or non-deterministic); the third column shows the number of lines of code (LOC); the fourth column provides the total number of properties identified, which is equivalent to the number of test cases that are created; and finally, the fifth column shows the number of properties violated. The adopted benchmarks were EUREKA[4], SNU[5], WCET[6], and a fragment of code extracted from the pulse oximeter device [Cordeiro et al. 2009a]. The results of the application of the proposed method are available at https://sites.google.com/site/fortesmethod/.

[4]www.ai-lab.it/eureka/bmc.html

[5]http://archi.snu.ac.kr/realtime/benchmark

[6]http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

Table 1. Details related to the benchmarks of C code used.

Number	Module	LOC	Identified	Violated
1	EUREKA_bf20_det.c	49	33	0
2	EUREKA_Prim_4_det.c	78	30	0
3	SNU_bs_nondet.c	120	7	0
4	SNU_crc_det.c	125	15	0
5	SNU_insertsort_nondet.c	94	14	6
6	SNU_qurt_det.c	164	6	0
7	SNU_qsort-exam_det.c	134	49	-
8	SNU_select_det.c	122	39	6
9	WCET_cnt_nondet.c	139	16	0
10	Oximeter_log_det.c	177	4	2

As presented before, the proposed method adopts ESBMC just for finding properties (in this context called *claims*), that may be violated, where such properties are then verified by the CUnit. Note that in this method, ESBMC is not used for properties verification. However, for validation purposes, ESBMC has been choosen to be also adopted as a validator, i.e. the proposed method should be compared with properties verification of ESBMC and, if all is well, to find the same violated properties that ESBMC alone (that is, without CUnit) can find. In Table 1, in lines from 1 to 4, 6 and 9 the respective codes have not violated any property in both the proposed method and ESBMC alone; in lines 8 and 10 were identified violation of 6 and 2 properties, respectively identified by the method and confirmed by ESBMC alone. However, there were two special cases, (i) in line 7, the method does not identify any errors, but ESBMC alone generated either `time out` or `out of memory`, in this case we cannot say anything about the verification; (ii) in line 5, the proposed method does not find any errors, however ESBMC identified violation of 6 properties. These issues will be addressed in future works.

The proposed method for identifying and verifying errors considering safety properties has demonstrated that this task is not trivial to be implemented without the adoption of a systematic methodology mainly because properties identification alone is not sufficient to guarantee that such property is violated. In order to detail further this point we adopted the code "Oximeter_log_det.c" of the pulse oximeter implementation [Cordeiro et al. 2009a] depicted in line 10 of Table 1. We can see that the ESBMC identified four properties. However, we will focus on a specific code fragment as shown in Figure 9, where ESBMC identified the following two properties: "`next` $<$ `6400`" in line 2, and "`(unsigned int)buffer_size != 0`" in line 3. We found out that this code fragment has properties that may be violated, resulting in problems of buffer overflow and division by zero. These problems can occur, because the variables "`buffer_size`" and "`next`" are set by another function, in this case "`void initLog(Data8 max)`". Hence, if someone does not set (or constrain) the value of "`buffer_size`" and "`next`", then the SMT solver can assign a non-deterministic value to the variables "`buffer_size`" and "`next`", for example, zero for "`buffer_size`" (which would cause a division by zero) or a value greater than "`BUFFER_MAX`", where this constant sets the size of the array, for "`next`" (which would cause a buffer overflow). Thus, the identification of possible errors sometimes need to explore other parts of the

code which, depending on the analyzed code, may become a very difficult task.

```
1 void insertLogElement(Data8 b){
2         buffer[next] = b;
3         next = (next+1)%buffer_size;
4 }
```

Figure 9. A fragment of the Oximeter_log_det.c code.

The assignment of values was made based on the possible problems identified above, that is: function "`initLog()`" is not called and function "`insertElementLog`" runs several times until variable "`next`" becomes greater than "`BUFFER_MAX`". The result of this verification is shown in Figure 10, where we note for the first situation, the *assert* statement is executed 2 times, while for the second situation, the *assert* statement is executed 12801 times before the property (or *assert*) is violated.

(i) first situation

```
Suite: log
  Test: testClaims ... FAILED
    1. log_CUnit.c:118  - (unsigned int)buffer_size != 0
--Run Summary: Type      Total     Ran  Passed  Failed
               suites        1       1     n/a       0
               tests         1       1       0       1
               asserts       2       2       1       1
```

(ii) second situation

```
Suite: log
  Test: testClaims ... FAILED
    1. log_CUnit.c:113  - next < 6400
--Run Summary: Type      Total     Ran  Passed  Failed
               suites        1       1     n/a       0
               tests         1       1       0       1
               asserts   12801   12801   12800       1
```

Figure 10. Result of verification performed with the method in the log.c code.

Abstraction of data and safety properties usually does not require a big effort. However, if the number of LOCs is high or properties identified is large, then this task may become very difficult. The implementation of the proposed method not only serve as a basis for the abstraction of the safety properties generated by model checkers, but also for the basic model for test engineers who are already familiar with the testing framework tools. The automation of this method can thus be of great advantage.

6. Conclusions and Future Work

In this work we presented a new method to integrate the model checker ESBMC with the CUnit framework in order to create and execute the test cases of sequential C programs. In particular, this method aims to extract the safety properties generated by ESBMC to automatically create test cases using the rich set of assertions provided by the CUnit test framework. The experimental results, although preliminary, have shown to be very effective over several publicly available benchmarks if compared to software model checking alone. Note also that with the set of tests performed, we were able to identify some improvements that will be implemented in the proposed method w.r.t. treatment and abstraction of information generated by the model checker (and safety properties resp.), such

as the code structure and block delimiters, and analysis of properties related to pointer and dynamic memory allocation. For future work, we intend to investigate the application of verification techniques with instrumented code [Nethercote and Seward 2007] and other approaches, such as the running code on symbolic inputs [Cadar and Engler 2005] and mutation testing [Jia and Harman 2010] that will allow us to improve the proposed method by enabling the generation of test cases in a wider range of properties.

Acknowledgement The authors acknowledge the support granted by FAPESP process 08/57870-9, and by CNPq processes 554071/2006-1, 575696/2008-7, and 573963/2008-8.

References

Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. MIT Press.

Barnett, M., Grieskamp, W., Gurevich, Y., Schulte, W., Tillmann, N., and Veanes, M. (2003). Scenario-oriented modeling in asml and its instrumentation for testing. In *Testing, 2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*.

Beyer, D., Henzinger, T., Jhala, R., and Majumdar, R. (2007). The software model checker BLAST: Applications to software engineering. In *International Journal on Software Tools for Technology Transfer (STTT)*, pages 505–525.

Borges, Rafael Magalh a. and Mota, A. C. (2007). Integrating uml and formal methods. In *Electron. Notes Theor. Comput. Sci.*, pages 97–112.

Cadar, C. and Engler, D. (2005). Execution generated test cases: How to make systems code crash itself. In *12th International SPIN Workshop on Model Checking of Software*.

Cordeiro, L., Fischer, B., Chen, H., and Marques-Silva, J. (2009a). Semiformal verification of embedded software in medical devices considering stringent hardware constraints. In *International Conference on Embedded Software and Systems*, pages 396–403.

Cordeiro, L., Fischer, B., and Marques-Silva, J. (2009b). SMT-based bounded model checking for embedded ANSI-C software. In *Automated Software Engineering*, pages 137–148.

Jia, Y. and Harman, M. (2010). An analysis and survey of the development of mutation testing. In *IEEE Transactions on Software Engineering*.

Marre, B. and Arnould, A. (2000). Test sequences generation from lustre descriptions: Gatel. In *15th IEEE international conference on Automated Software Engineering*, page 229.

Myers, G. J. and Sandler, C. (2004). *The Art of Software Testing*. John Wiley & Sons.

Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM on Programming Language Design and Implementation*, pages 89–100.

Petrenko and Alexandre (2001). Fault model-driven test derivation from finite state models: Annotated bibliography. In *Modeling and Verification of Parallel Processes*, pages 196–205.

Schmitt, M., Ebner, M., , and Grabowski, J. (2000). Test generation with autolink and testcomposer. In *2nd Workshop of the SDL Forum Society on SDL and MSC*, pages 26–28.