UNIVERSIDADE FEDERAL DO AMAZONAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

EXPLOITING SAFETY PROPERTIES IN BOUNDED MODEL CHECKING FOR TEST CASES GENERATION OF C PROGRAMS

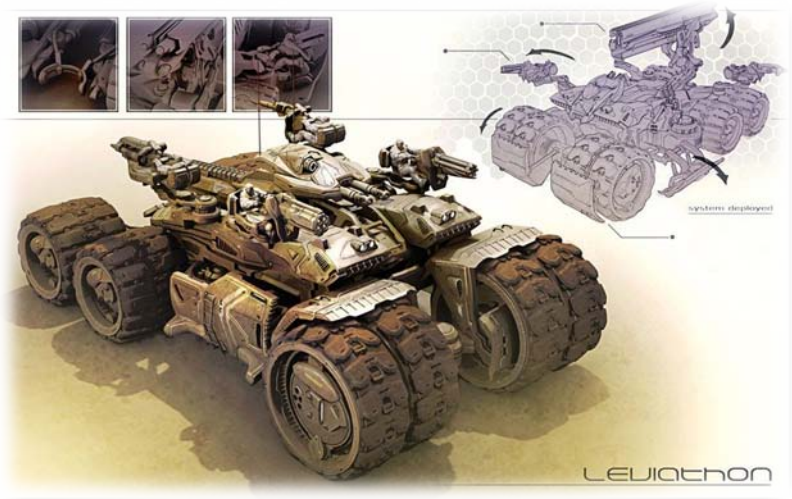
Herbert Oliveira Rocha, Lucas Cordeiro
Raimundo Barreto and José Netto

SAST'10

Outline

1. Introduction
2. Background
3. Related Work
4. Proposed Method
5. Experimental Results
6. Conclusions and Future Work
7. Questions?

Software Applications



Model Checking

Model Checking

- Area of research in formal verification;
- Software categories.

The main challenges

- State space explosion problem;
- Integration with test environments;

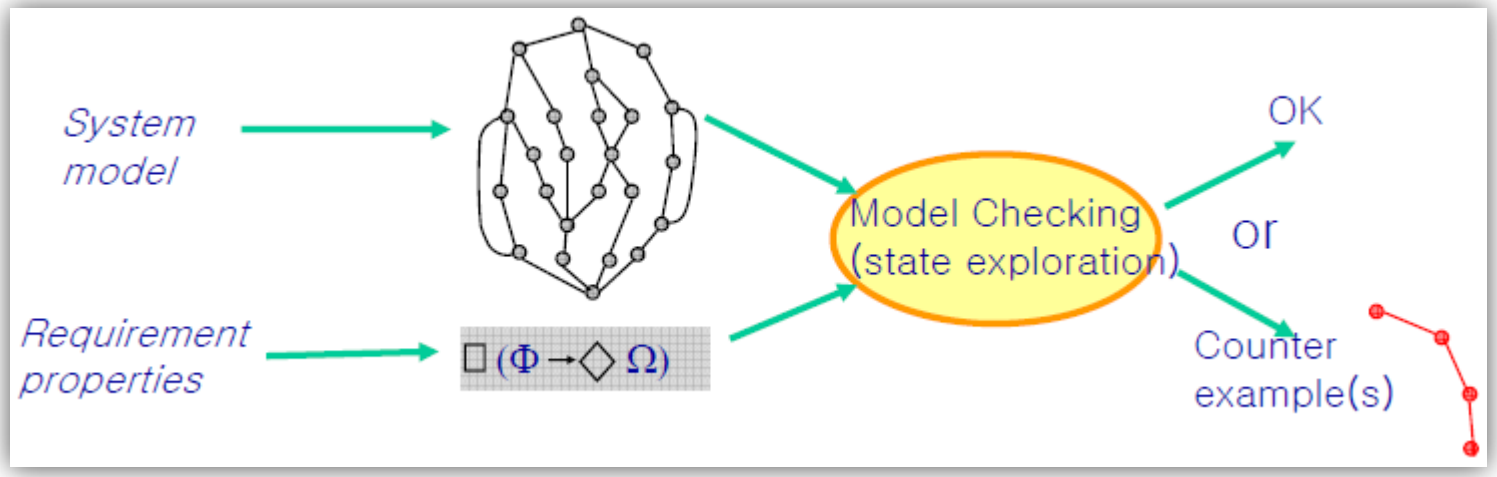
And what are we proposing?

- According to [Baier and Katoen 2008][1] safety properties are often characterized as “nothing bad should happen”.

1- Principles of Model Checking. MIT Press. 2008

What is Model Checking?

The procedure normally generates an exhaustive search in the state space model to determine whether a given “property” is valid or not [Baier and Katoen 2008][2].



2- Principles of Model Checking. MIT Press. 2008

Efficient SMT-Based Bounded Model Checker (ESBMC)

ESBMC[3] is a bounded model checker for embedded ANSI-C software based on SMT (Satisfiability Modulo Theories) solvers, which allows [Cordeiro et al. 2009] [4]:

- ✓ Out-of-bounds array indexing;
- ✓ Division by zero;
- ✓ Pointers safety
- ✓ Dynamic memory allocation;
- ✓ Data races;
- ✓ Deadlocks;
- ✓ Unwinding of the loops;
- ✓ Underflow e Overflow;
- ✓ Softwares Multi-threaded.

3- http://users.ecs.soton.ac.uk/lcc08r/esbmc/
4- SMT-Based BoundedModel Checking for Embedded ANSI-C Software. ASE 2009.

Exploting Safety Properties in Software Testing Strategies

What is software testing?

- Software testing is the process of executing a program with the goal of finding faults [Myers and Sandler 2004][5]

The CUnit Framework

The problem is "***how to creat test cases aimed at checking safety properties?***"

5- The Art of Software Testing. John Wiley & Sons. 2004

Related Work

Test sequences generation from LUSTRE descriptions: GATEL. [Marre e Arnould, 2000][6]

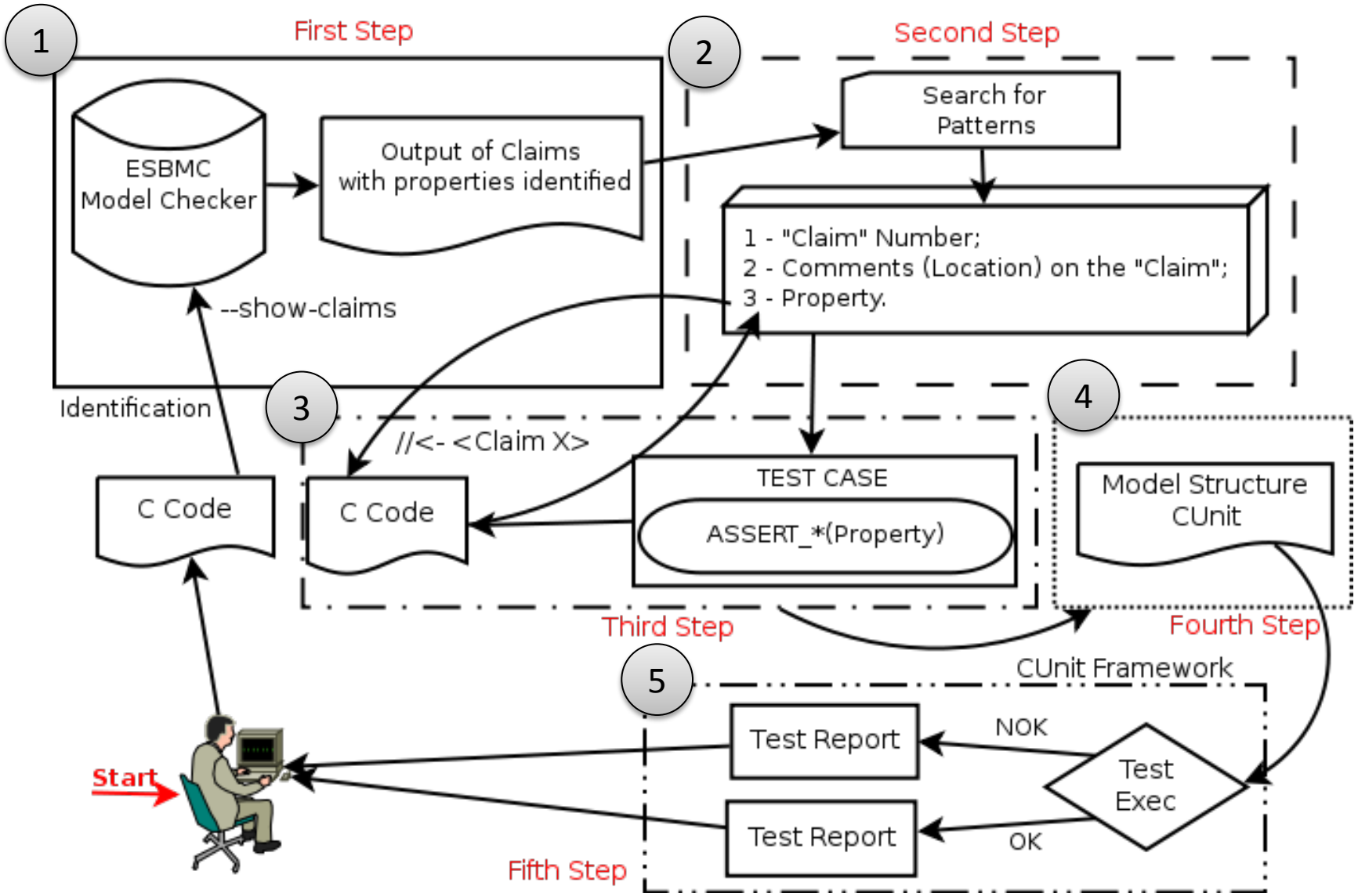
Scenario-oriented modeling in Asml and its instrumentation for testing. [Barnett et al. 2003][7]

Test generation with Autolink and Testcomposer. [Schmitt et al. 2000][8]

Execution generated test cases: How to make systems code crash itself. [Cadar and Engler 2005][9]

6- Test sequences generation from LUSTRE descriptions: GATEL. ASE. 2000
7- Scenario-oriented modeling in Asml and its instrumentation for testing. SCESM. 2003
8- Test generation with Autolink and Testcomposer. SAM. 2000
9- Execution generated test cases: How to make systems code crash itself. 12th MCS, Stanford Technical Report CSTR 2005-04. 2005

Proposed Method



In order to explain the main steps of the proposed…

```
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;
7       b[0] = 1;
8       temp = a[i]*(i+1);
9       for(j=1;j<temp;j++){ b[j] = b[i-1]+(temp*2); }
10  }
11 }
```

First step: Identification of Safety Properties

```
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
```

1
2
3

Result_ESBMC.txt

Second step: Safety Properties Information Collection

Result_ESBMC.txt

Output of ESBMC

file sum_array.c: Parsing
Converting
Type-checking sum_array
Generating GOTO Program
Pointer Analysis
Adding Pointer Checks
Claim 1:
Claim 2:
Claim 3:
Claim 4:
Claim 5:
Claim 6:
Claim 7:
Claim 8:
Claim 9:
Claim 10:

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

result_claims.txt

Third step: Asserts Inclusion

(i) First phase

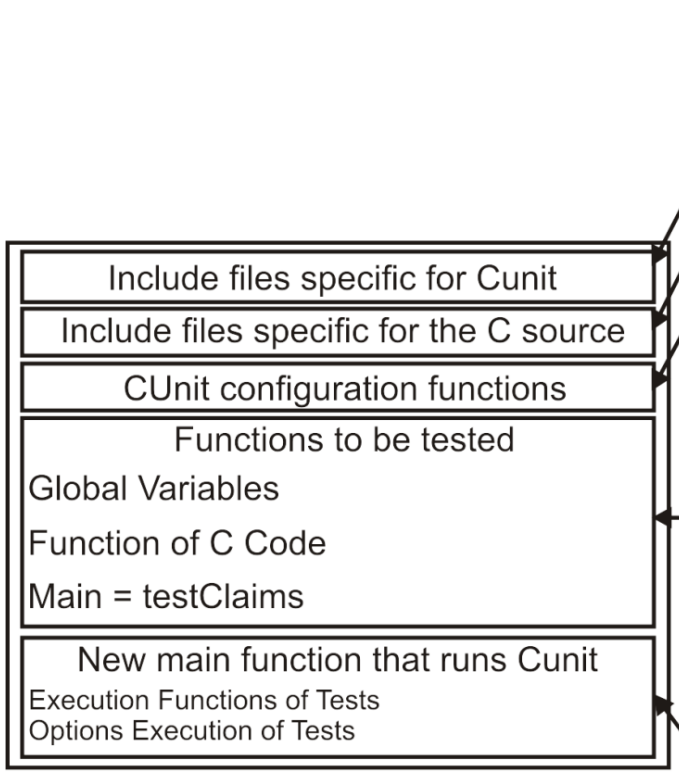
```
1 for( i=1; i<5; i++){
2   a[i]= a[i-1] + i; //<- <Claim 1> <Claim 2> <Claim 3> <Claim 4>
3   b[0] = 1;
4   temp = a[i]*(i+1); //<- <Claim 5> <Claim 6>
5   for(j=1;j<temp;j++){
6     b[j]=b[i-1]+(temp*2); //<- <Claim 7> <Claim 8> <Claim 9>
7   }
8 }
```

Third step: Asserts Inclusion

(ii) Second phase

```
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>
```

Fourth step: Implementing Unit Test in CUnit Framework




```
1.  #include "CUnit/Basic.h"
2.  #include <stdio.h>
3.  #include <string.h>
4.  int init_suite1(void) { ... }
5.  int clean_suite1(void) { ... }
6.  int a[5];
7.  int b[6];
8.  void testClaims(void) {
9.      int i, j, temp;   a[0] = 1;
10.     for (i = 1; i < 5; i++) {
11.         //Claim 1:
12.         //file sum_array.c line 9 function main
13.         //array `a' lower bound
14.         CU_ASSERT(i >= 0);
15.         a[i] = a[i - 1] + i; //<- Claim 1
16.        { ... }
17.     }
18. }
19. int main() {
20.  CU_pSuite pSuite = NULL;
21.  pSuite = CU_add_suite("sum_Array", init_suite1, clean_suite1);
22.  if (NULL == pSuite) { ... }
23.  if (CUE_SUCCESS != CU_initialize_registry()) return CU_get_error();
24.  if ((NULL == CU_add_test(pSuite, "testClaims", testClaims))) {...}
25.  CU_basic_set_mode(CU_BRM_VERBOSE);
26.  CU_basic_run_tests();    CU_cleanup_registry();
27.  return CU_get_error();
28. }
```

Fifth step: Running CUnit Tool

```
     CUnit - A Unit testing framework for C - Version 2.1-0
     http://cunit.sourceforge.net/
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
Pressione [Enter] para fechar o terminal...
```

Experimental Results

The steps of the proposed method have been implemented using the ESBMC v1.11, and the framework CUnit v2.1.

Number	Module	LOC	Identified	Violated
1	EUREKA_bf20_det.c[10]	49	33	0
2	EUREKA_Prim_4_det.c[10]	78	30	0
3	SNU_bs_nondet.c[11]	120	7	0
4	SNU_crc_det.c[11]	125	15	0
5	SNU_insertsort_nondet.c[11]	94	14	6
6	SNU_qurt_det.c[11]	164	6	0
7	SNU_qsort-exam_det.c[11]	134	49	-
8	SNU_select_det.c[11]	122	39	6
9	WCET_cnt_nondet.c[12]	139	16	0
10	Oximeter_log_det.c[13]	177	4	2

Codes are available at : https://sites.google.com/site/fortesmethod/

10- www.ai-lab.it/eureka/bmc.html
11- http://archi.snu.ac.kr/realtime/benchmark
12- http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
13- Semiformal verification of embedded software in medical devices considering stringent hardware constraints . ICESS 2009.

Identifying and verifying errors

This task is not trivial

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

A specific **Oximeter_log_det.c a** code fragment

void **initLog**(Data8 max)

Properties Identified:

1. next < 6400 -> **Line 2**
2. “(unsigned int)buffer size != 0 -> **Line 3**

Verification Results

First 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
```

Second 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
```

Conclusions and Future Work

Proposed Method

- The experimental results, although preliminary, have shown to be very effective;
- We identify some improvements;
  - About code structure and block delimiters;
  - About the verifying pointer and dynamic memory allocation.

Future work

- We intend to investigate the application of verification techniques, such as:
  - Running code on symbolic inputs [Cadar and Engler 2005][14];
  - Mutation testing[Jia and Harman 2010][15].

14- Execution generated test cases: How to make systems code crash itself. 12th MCS, Stanford Technical Report CSTR 2005-04

15- An analysis and survey of the development of mutation testing. IEEE TSE 2010

Questions ??

Thank you for your attention!