



Hunting Memory Bugs in C Programs with Map2Check (Competition Contribution)



Herbert Rocha, Raimundo Barreto,
and Lucas Cordeiro



TACAS 2016

Competition on Software Verification (SV-COMP)

Verification and Testing Software

In **software testing**:

- ✓ a significant human effort is required to generate effective test cases
- ✓ subtle bugs are difficult to detect

In **software model checking**:

- ✓ limited scalability to large software
- ✓ missing tool-supported integration into the development process

Verification and Testing Software

In **software testing**:

- ✓ a significant human effort is required to generate effective test cases
- ✓ subtle bugs are difficult to detect

In **software model checking**:

- ✓ limited scalability to large software
- ✓ missing tool-supported integration into the development process

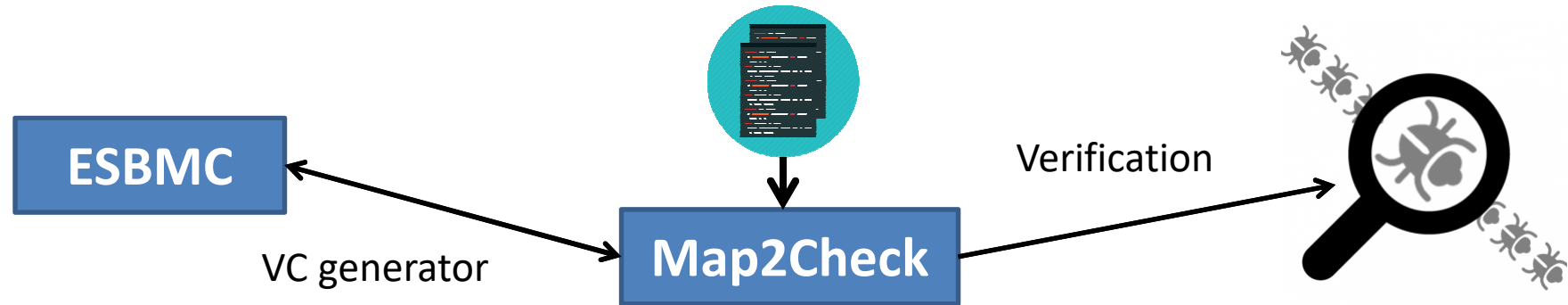
The integration aims to alleviate the weaknesses from those strategies

Hunting Memory Bugs with Map2Check

- ✓ Map2Check automatically generates and checks:
 - **memory management test cases** for structural unit tests for C programs
 - assertions from **safety properties** generated by **BMC tools**
- ✓ Map2Check aims to improve the unit testing environment, adopting features from (bounded) model checkers
- ✓ Map2Check adopts **source code instrumentation to:**
 - monitor the program's executions
 - validate assertions with **safety properties**

Hunting Memory Bugs with Map2Check

Map2Check method **checks** the program **out of the BMC tools flow**



- ✓ It is based on dynamic analysis and assertion verification
- ✓ The assertions contain a set of specifications
- ✓ The **BMC** is adopted as verification condition (VC) generator
- ✓ Checks for SV-COMP properties “invalid-free”, “invalid-deference”, and “memory-leak”

Hunting Memory Bugs with Map2Check

```
3.  int *a, *b;
4.  int n;
5.
6.  #define BLOCK_SIZE 128
7.
8.  void foo (){ ... }
16.
17.  int main ()
18.  {
19.      n = BLOCK_SIZE;
20.      a = malloc (n * sizeof(*a));
21.      b = malloc (n * sizeof(*b));
22.      *b++ = 0;
23.      foo ();
24.      if (b[-1])
25.      { /* invalid free (b was iterated) */
26.          free(a); free(b); }
27.      else
28.      { free(a); free(b); } /* ditto */
29.
30.      return 0;
31.  }
```

960521-1_false-valid-free.c

SV-COMP 2016

Step 1: Identification of safety properties

```
$ esbmc --64 --no-library --show-claims
960521-1_false-valid-free.c
file 960521-1_false-valid-free.c: Parsing
Converting
Type-checking 960521-1_false-valid-free
Generating GOTO Program
Pointer Analysis
Adding Pointer Checks
Claim 1:
file 960521-1_false-valid-free.c line 12 function foo
dereference failure: dynamic object lower bound
```

Dereferencing
operation is a dynamic
object

```
!(POINTER_OFFSET(a) + i < 0) || !(IS_DYNAMIC_OBJECT(a))
```

Claims generated automatically by ESBMC do not necessarily correspond to errors

Step 2: Extract information from safety properties

Claims	Comments	Line	Property
Claim 1	dereference failure: dynamic object lower bound	12	$\text{!(POINTER_OFFSET}(a) + i < 0)$ $\text{ !(IS_DYNAMIC_OBJECT}(a))$
Claim 2	dereference failure: dynamic object upper bound	12	$\text{!(POINTER_OFFSET}(a) + i \geq$ $\text{DYNAMIC_SIZE}(a)) \text{ }$ $\text{!(IS_DYNAMIC_OBJECT}(a))$
Claim 3	dereference failure: dynamic object lower bound	14	$\text{!(POINTER_OFFSET}(b) + i < 0)$ $\text{ !(IS_DYNAMIC_OBJECT}(b))$
Claim 4	File sum_array line 14 function main array `a` upper bound	14	$\text{!(POINTER_OFFSET}(b) + i \geq$ $\text{DYNAMIC_SIZE}(b)) \text{ }$ $\text{!(IS_DYNAMIC_OBJECT}(b))$
...

Step 3: Translation of safety properties

Translate claims provided by ESBMC into assertions written in C code:

✓ INVALID-POINTER.

INVALID - POINTER(*i + pat*) *to*
`IS_VALID_POINTER_MF (LIST_LOG, (void*)&(i+pat), (void*)(intptr_t)(i+pat))`



`#include <map2check.h>`

Map2Check provides a **library** to the C program, which offers support to execute the functions generated by the translator.

Step 4: Memory tracking

```
3. int *a, *b;
4. int n;
5.
6. #define BLOCK_SIZE 128
7.
8. void foo () { ...
16.
17. int main ()
18. {
19.     n = BLOCK_SIZE;
20.     a = malloc (n * sizeof(*a));
21.     b = malloc (n * sizeof(*b));
22.     *b++ = 0;
23.     foo ();
24. }
```

Phase 1: identify and track variables

Input: Abstract Syntax Tree (AST)

Output: Variables Tracking (Map)

Analyzing the
program scope

foreach *node* **IN** the AST **do**

if *type(node)* == *FuncDef* **then**

compound_func = get the sub tree from node

foreach *subNo* **FROM** *compound_func* == *Decl* **do** *getDataFromVar(subNo, 0)* ;

end

else if *type(node)* == *Decl* **then** *getDataFromVar(node, 1)* ;

end

Function *getDataFromVar(node, enableGlobalSearch)*

Step 4: Memory tracking

Tracking of the variables

```
3. int *a, *b;
4. int n;
5.
6. #define BLOCK_SIZE 128
7.
8. void foo (){ ... }
16.
17. int main ()
18. {
19.     n = BLOCK_SIZE;
20.     a = malloc (n * sizeof(*a));
21.     b = malloc (n * sizeof(*b));
22.     *b++ = 0;
23.     foo ();
24.     if (b[-1])
25.     { /* invalid free (b was iterated) */
26.     free(a); free(b); }
27.     else
28.     { free(a); free(b); } /* ditto */
29.
30.     return 0;
31. }
```

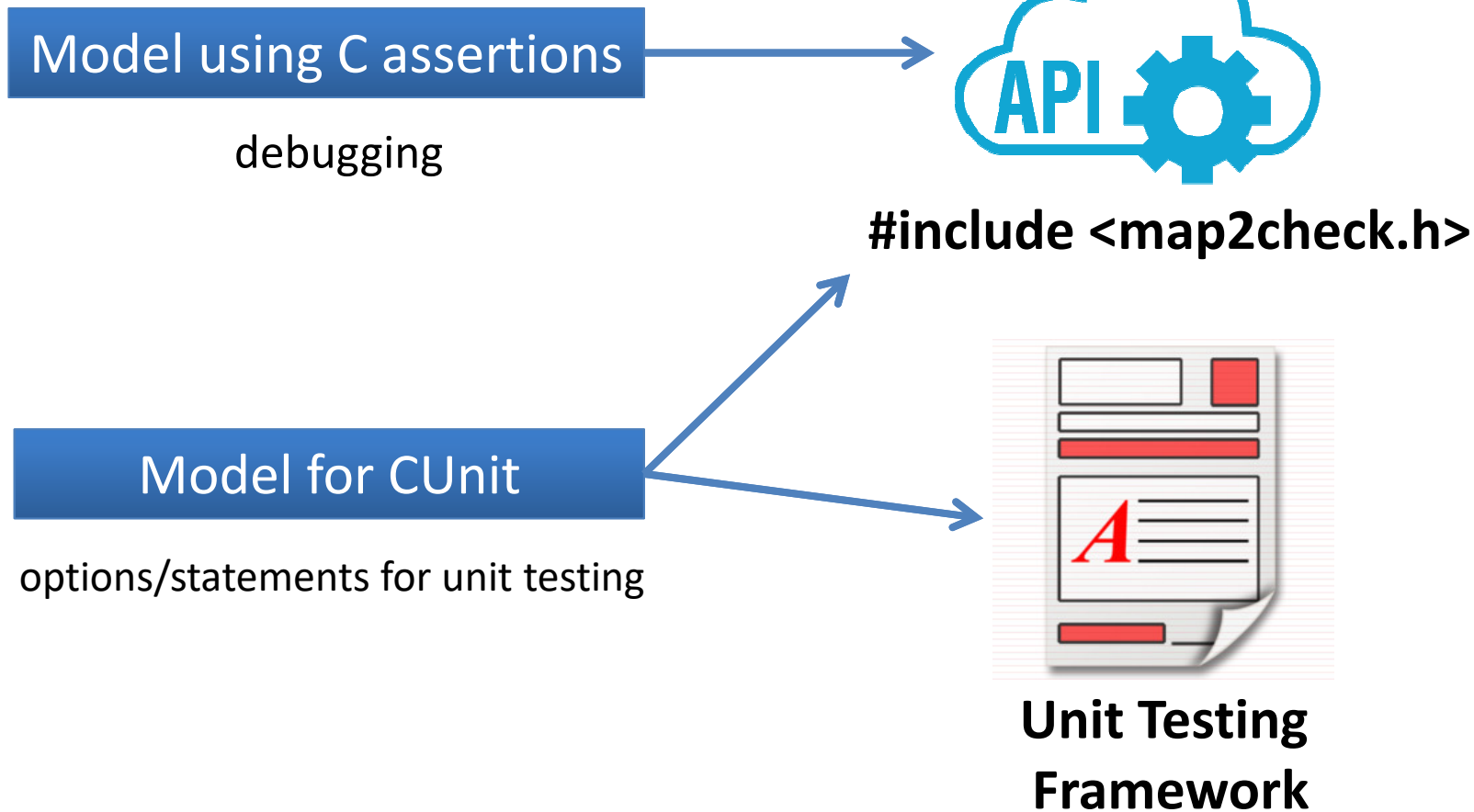
Pointer variable
assignments



Step 5: Code instrumentation with assertions

```
16. ...
17.
18. int main ()
19. {
20.     n = BLOCK_SIZE;
21.     a = malloc (n * sizeof(*a));
22.     b = malloc (n * sizeof(*b));
23.     *b++ = 0;
24.     foo ();
25.     if (b[-1])
26.     {
27.         ...
28.     }
29.     else
30.     {
31.         ASSERT(INVALID_FREE(LIST_LOG, (void *) (intptr_t) (a), 28));
32.         free(a);
33.         ASSERT(INVALID_FREE(LIST_LOG, (void *) (intptr_t) (b), 28));
34.         free(b);
35.     }
36.     return 0;
37. }
```

Step 6: Implementation of the tests



Step 7: Execution of the tests

Tracking memory execution

```
3. int *a, *b;
4. int n;
5.
6. #define BLOCK_SIZE 128
7.
8. void foo (
16.
17. int
18. {
19.     n
20.     a
21.     b
22.     *
23.     f
24.     if (n-1)
25.     {
26.     f
27.     else
28.     { free(a); free(b); } /* ditto */
29.
30.     return 0;
31. }
```

Invalid free

Line	Address	Points to	Escape	Is Dynamic	Is Free
28	0x601050	0xb44034	global	0	1
28	0x601060	0xb44010	global	0	1
...
10	0x7fff39f18a2c	(nil)	foo	0	0
22	0x601050	0xb44034	global	0	0
21	0x601050	0xb44030	global	1	0
...

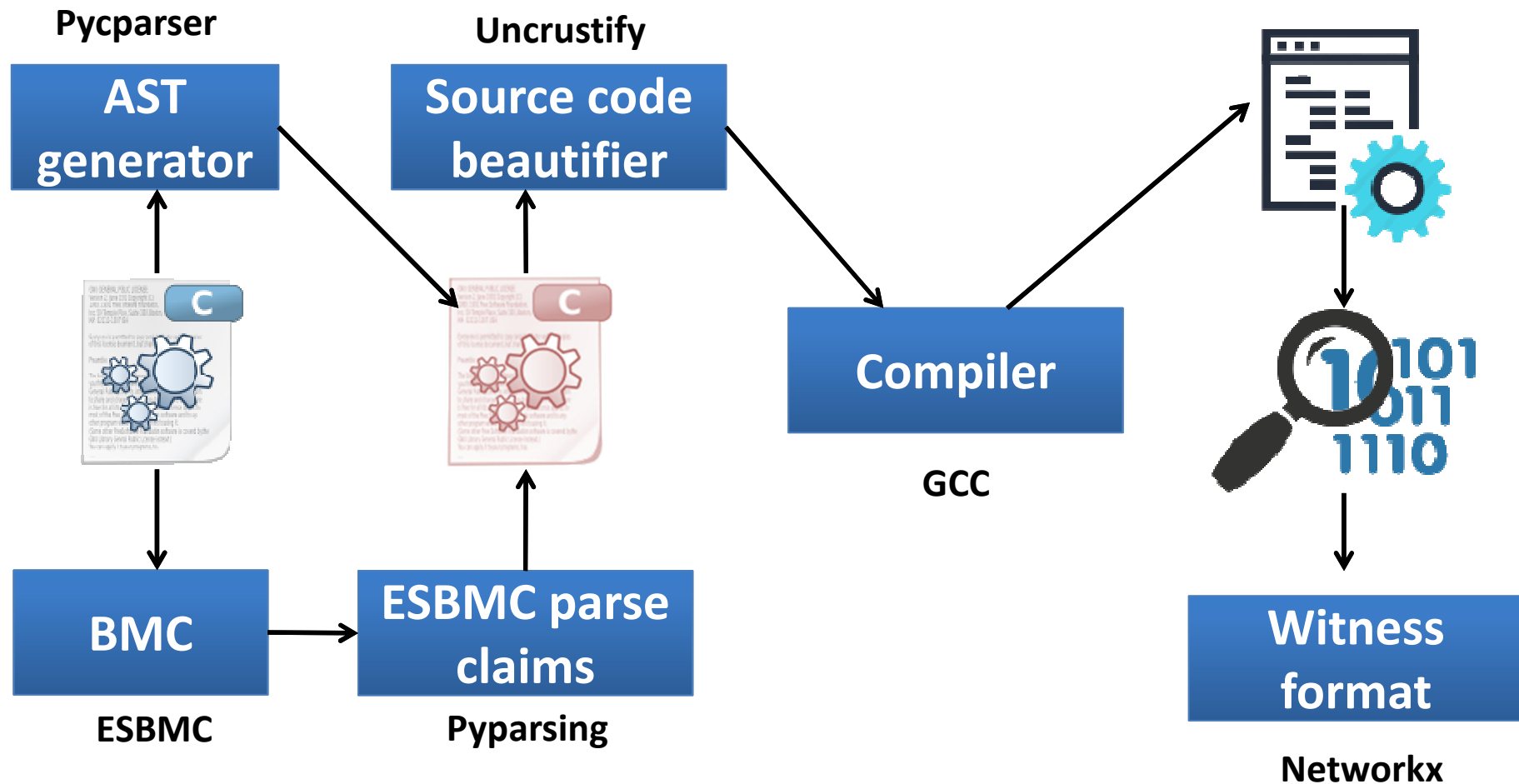
variable b was iterated

Strengths and Weaknesses - Map2Check

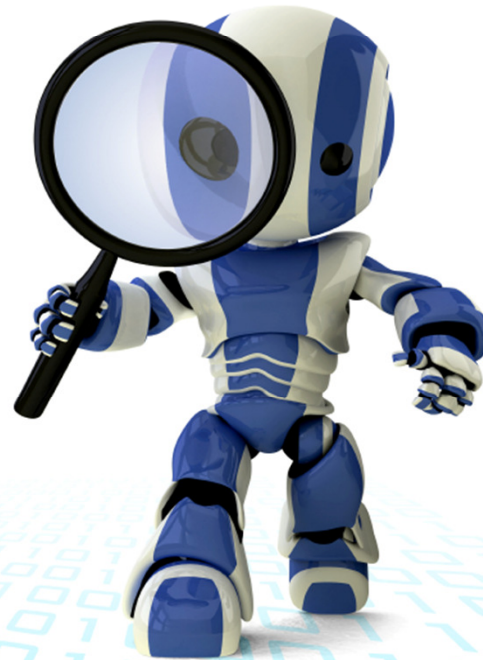
- ✓ Map2Check participates in the Heap Data Structures category only.
- ✓ The strength of the **tool lies in the precision** of its answers based on the **concrete execution of the analyzed program** over the VCs generated by ESBMC
- ✓ In preliminary experiments, Map2Check outperforms ESBMC due to timeouts or memory model limitations.
- ✓ The strategy based on **random data to unwind loops** and their respective loop exit condition do **not allow the correct execution of the program.**

Architecture, Implementation and Availability

source-to-source transformation



Map2Check tool is available at <https://github.com/hbgit/Map2Check>



Thank you for your attention!

map2check.tool@gmail.com