

# ECBS 2013

## SMT-Bounded Model Checking of C++ Programs

Mikhail Ramalho, **Mauro Freitas**, Felipe Sousa,  
Hendrio Marques, Lucas Cordeiro, Bernd Fischer



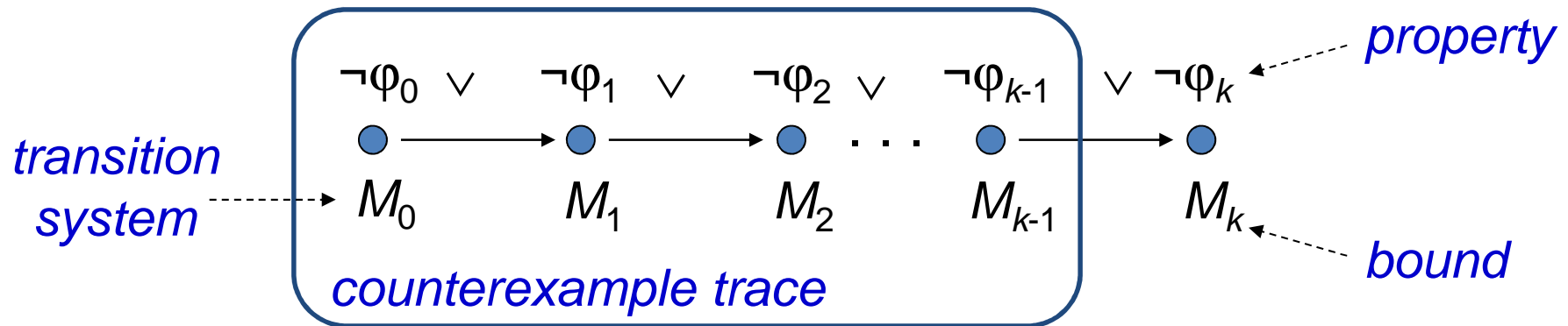
**UFAM**

UNIVERSITY OF  
**Southampton**  
School of Electronics  
and Computer Science



# Bounded Model Checking (BMC)

Idea: check negation of given **property** up to given **depth**



- transition system  $M$  unrolled  $k$  times
  - for programs: unroll loops, unfold arrays, ...
- translated into verification condition  $\psi$  such that
  - $\psi$  **satisfiable iff  $\varphi$  has counterexample of max. depth  $k$**
- has been applied successfully to verify (sequential) software

# BMC of C++ Programs

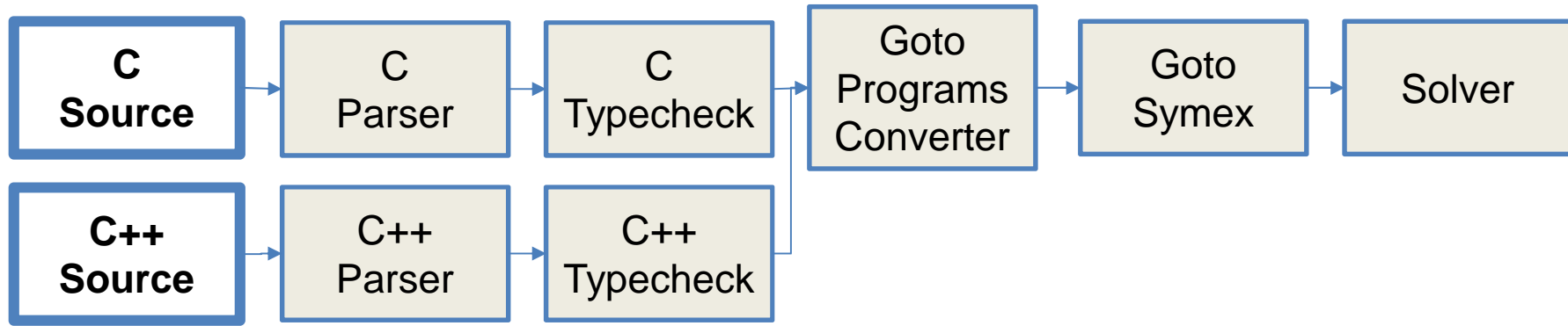
- there have been attempts to apply BMC to the verification of C++ programs but with **limited success**
  - handle **large programs** and **support complex features**
- problem: BMC of C++ programs presents greater challenges than that of C programs
  - more complex features such as **templates**, **containers**, and **exception handling** (contains and handles error situations in embedded systems)
- main insights:
  - optimized implementation of the standard C++ library complicates the VCs unnecessarily
  - abstract representation of the standard C++ libraries to conservatively approximate their semantics

# Objective of this work

## Extend BMC to support complex features of C++

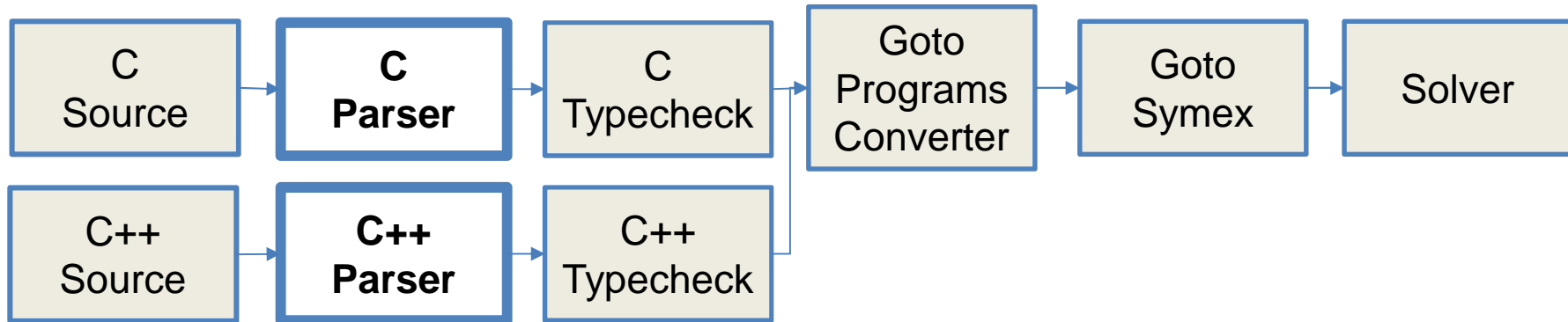
- exploit background theories of Satisfiability Modulo Theories (SMT) solvers
- provide suitable encodings for
  - template
  - exception handling
  - containers
  - arithmetic over- and underflow
- build and evaluate an SMT-based BMC tool (ESBMC++)
  - build on top of CBMC front-end
  - use different SMT encodings as back-ends

# ESBMC Architecture (1)



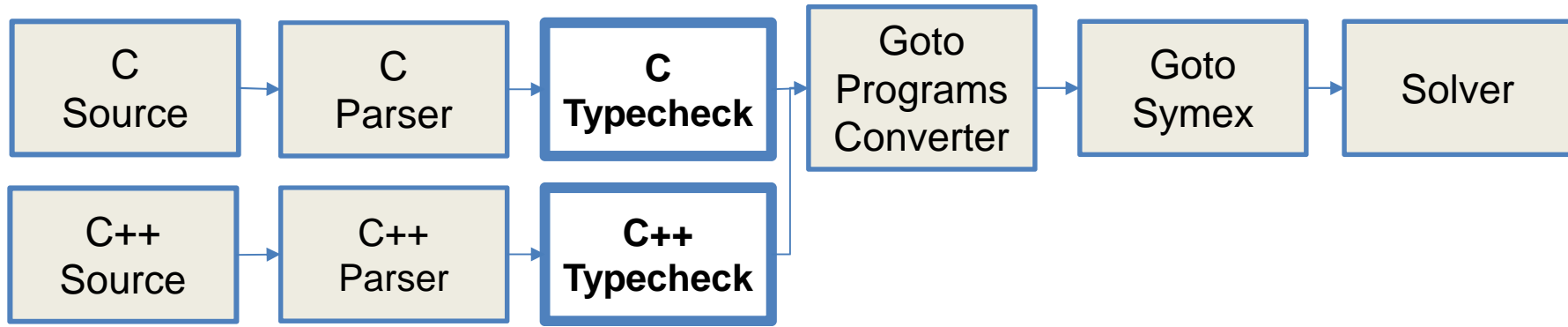
- originally only ANSI-C language was supported
- extend to support the verification of C++ programs with:
  - template (creation and instantiation)
  - exception handling (converted to goto functions)
  - standart template library (operational model)

# ESBMC Architecture (2)



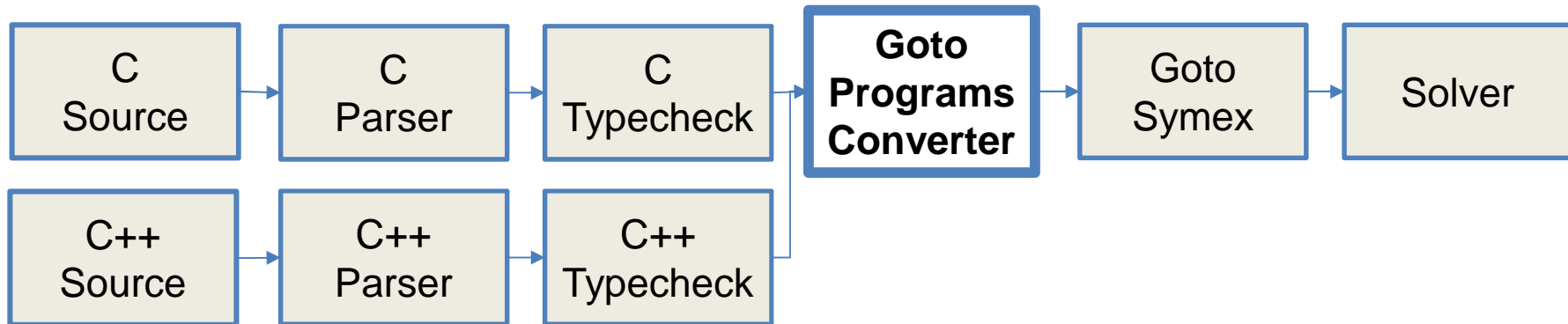
- lexer/parser based on the flex/bison
- most of the intermediate representation of the program (IRep) is created
  - this IRep is the base for the remaining phases of the verification

# ESBMC Architecture (3)

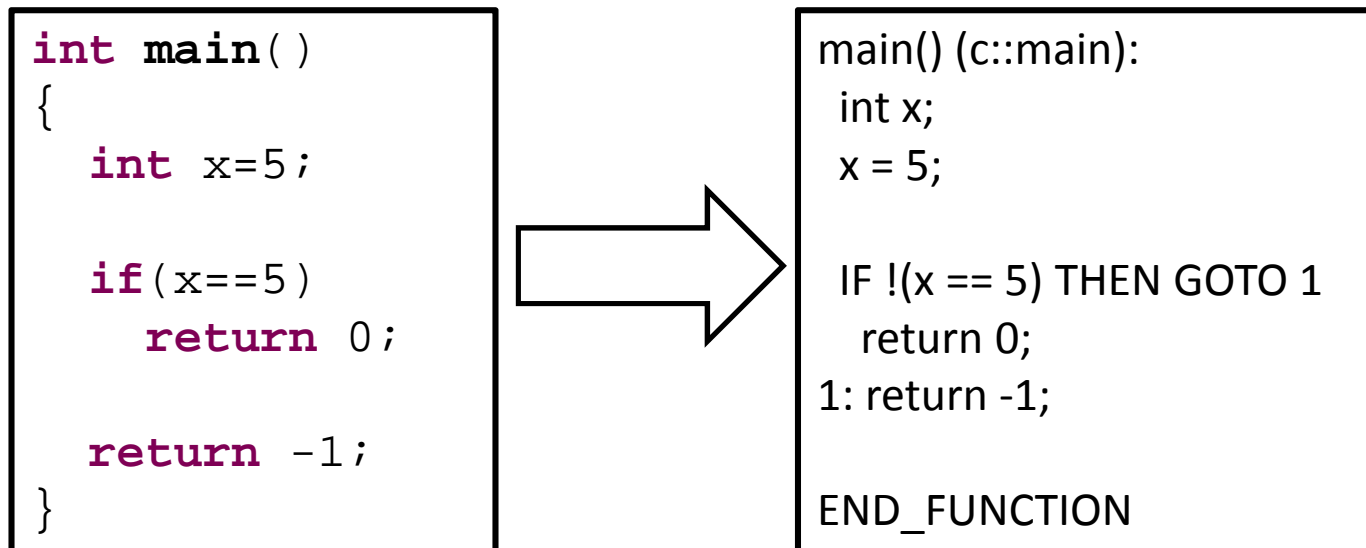


- some checks are made in this step:
  - assignment check
  - typecast check
  - pointer initialization check
  - function call check
  - template instantiation

# ESBMC Architecture (4)

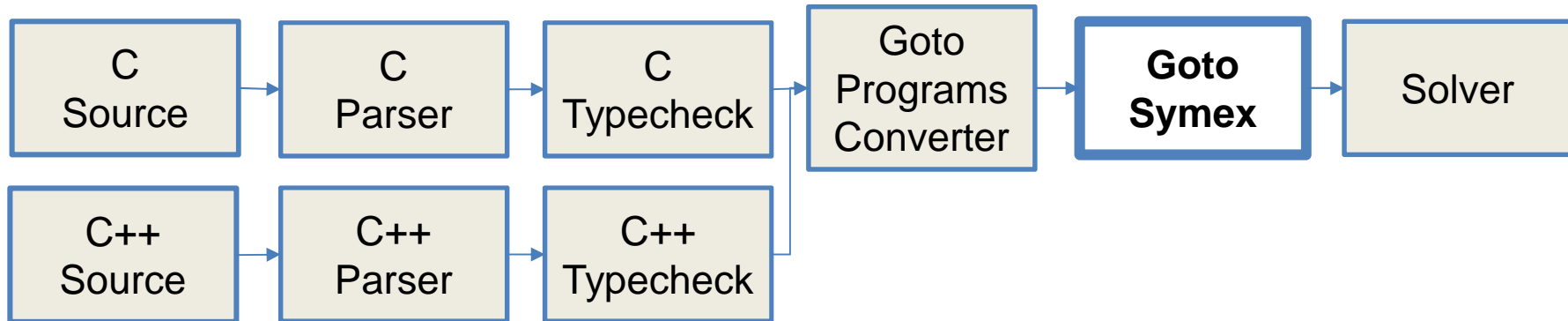


- conversion from **IRep** to **goto programs**:





# ESBMC Architecture (5)



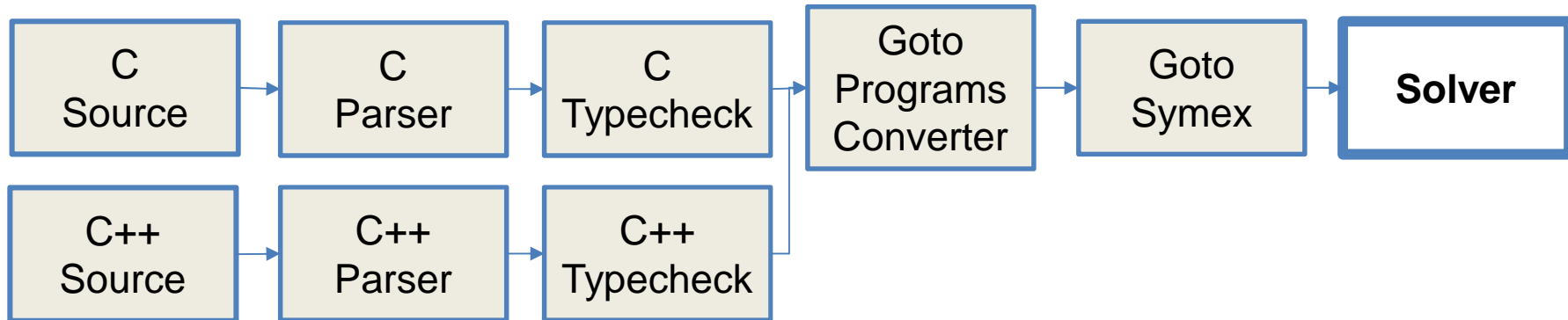
- creation of SSA expressions from goto programs:
  - assertions are inserted to check for pointer safety, memory-leak, division by zero, etc
  - **jump** instructions are inserted for **exception handling**

```
x = 5;  
x = 6;  
y = x;
```



```
x1 = 5;  
x2 = 6;  
y1 = x2;
```

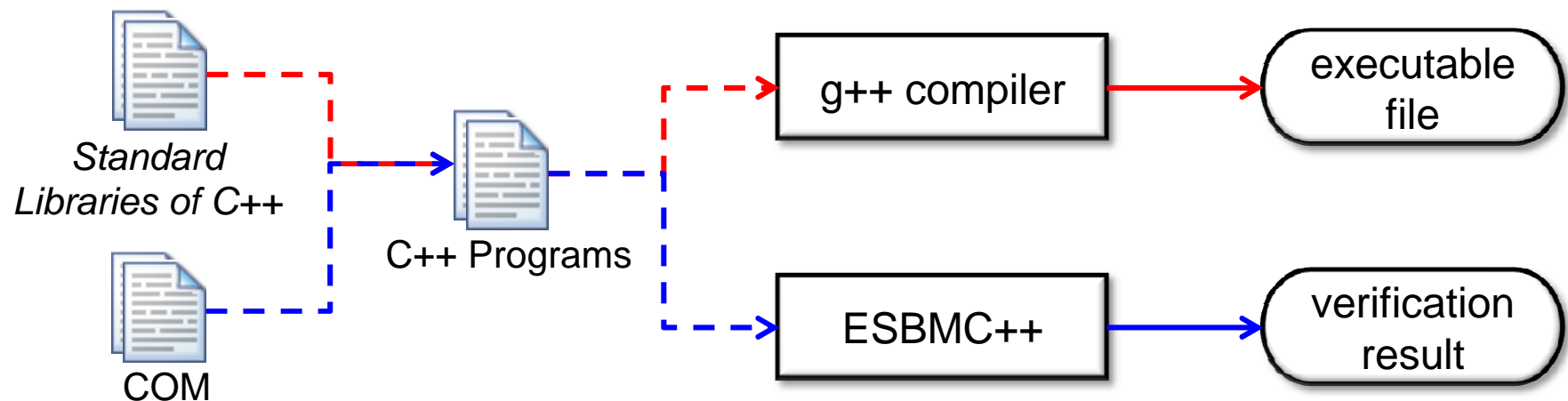
# ESBMC Architecture (6)



- encoding to bit-vector or integer/real arithmetic
- verification results can depend on encodings:
  - majority of VCs solved faster if numeric types are modelled by abstract domains but possible loss of precision

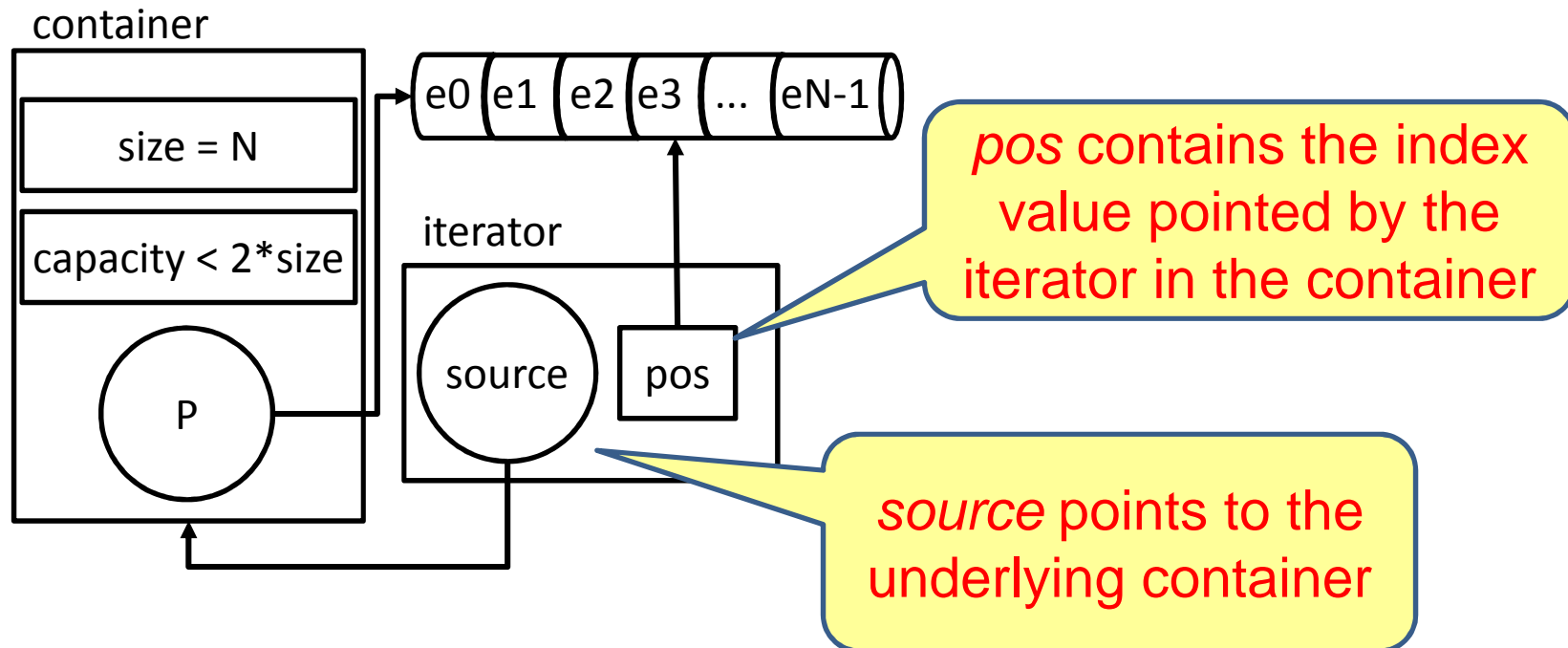
# SMT-Based BMC of C++ Programs

- there have been attempts to apply BMC to the verification of C++ programs but with **limited success**
  - handle **large programs** and **support complex features**
- standard C++ libraries contain complex (and low-level) data structures (complicates the VCs unnecessarily)
  - provide a C++ operational model (COM) which is an **abstract representation** of the standard C++ libraries that **conservatively approximates** their semantics



# Container Model (1)

- the **container model** uses three variables:
  - **P** that points to the first element of the array
  - **size** that stores the quantity of elements in the container
  - **capacity** that stores the total capacity of a container
- **iterators** are modelled using two variables (*source* and *pos*)



## Container Model (2)

- the core container model only supports the **insert**, **erase**, and **search** methods
  - `push_back`, `pop_back`, `front`, `back`, `push_front`, and `pop_front` are variations of these basic methods

$C((c', i') = c.erase(i)) :=$

$\wedge c'.size = c.size - 1$

$\wedge c'.array = store(...(store(c.array,$   
 $i.pos, select(c.array, i.pos + 1)),$   
 $...,$   
 $c.size - 2, select(c.array, c.size - 1))$

$\wedge i'.source = c'$

$\wedge i'.pos = i.pos$

decrement the size  
of the container

the exclusion is made  
by a given position,  
regardless the value

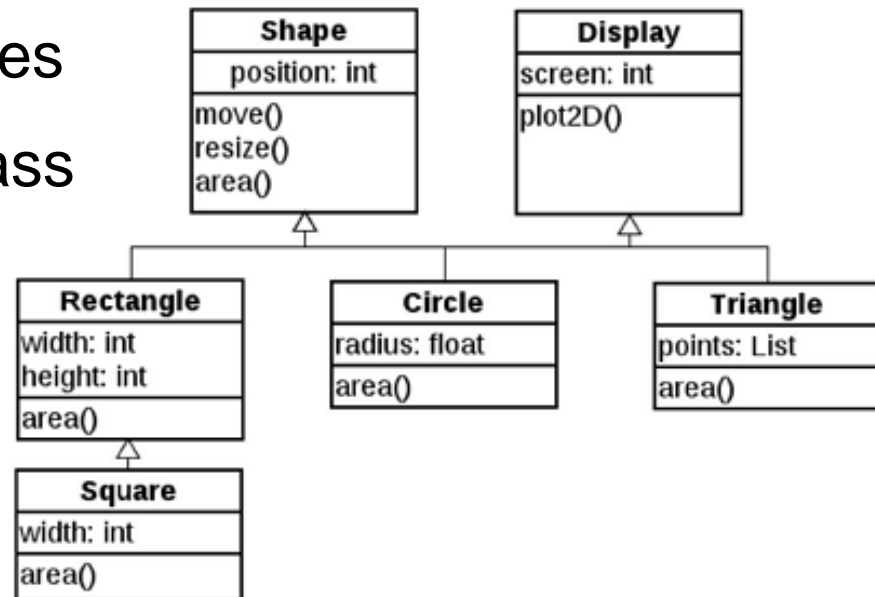
points to the position next  
to the previously erased  
part of the container

# Inheritance and Polymorphism

- polymorphism allows the creation of reusable code by changing only specific methods from the base class
  - in contrast to Java, C++ allows **multiple inheritance** which increase the complexity of the static analysis
- in ESBMC++, each new class instantiation replicate all the methods and attributes from the base classes
  - this feature allows base classes pointers to keep reference to derived classes
  - during verification time decides which method is being called from such pointer

# Running Example (1)

- triple  $\langle C, \prec_s, \prec_r \rangle$  where  $C$  is the set of classes
  - shared inheritance  $\prec_s \subseteq C \times C$
  - replicated inheritance  $\prec_r \subseteq C \times C$
- square class relation:  $\langle C, \emptyset, \{(Square, Rectangle, Shape), (Square, Rectangle, Display)\} \rangle$ 
  - direct access to the attributes and methods of the derived class
  - replicate information to any new class



# Running Example (2)

```
Square (int w) : Rectangle(w,w)
{ width = w; }

int area(void) { return width*width; }
```

Square  
constructor and  
area method

```
Square(10);
== 100);
```

$$C := \left[ \begin{array}{l} j_1 = store(j_0, vtable, Rectanle) \\ \wedge j_2 = store(j_1, width, 10) \\ \wedge j_3 = store(j_2, height, 10) \\ \wedge j_4 = store(j_3, vtable, Square) \\ \wedge j_5 = store(j_4, width, 10) \\ \wedge return\_value_1 = \\ (select(j_5, width) \times select(j_5, width)) \end{array} \right]$$
$$P := [return\_value_1 = 100]$$



# Running Example (2)

```
Square (int w) :  
{ width = w; }  
  
int area(void) { return width*width; }
```

Instantiation of  
square and area  
call

```
Shape *sqre = new Square(10);  
assert (sqre->area() == 100);
```

$$C := \left[ \begin{array}{l} j_1 = store(j_0, vtable, Rectanle) \\ \wedge j_2 = store(j_1, width, 10) \\ \wedge j_3 = store(j_2, height, 10) \\ \wedge j_4 = store(j_3, vtable, Square) \\ \wedge j_5 = store(j_4, width, 10) \\ \wedge return\_value_1 = \\ (select(j_5, width) \times select(j_5, width)) \end{array} \right]$$
$$P := [return\_value_1 = 100]$$

# Running Example (2)

```
Square (int w) : Rectangle(w,w)
{ width = w; }

int area(void) { return width*width; }
```

```
Shape *sqre = new Square(10);
assert (sq
```

Internal SMT  
representation

$$C := \left[ \begin{array}{l} j_1 = \text{store}(j_0, \text{vtable}, \text{Rectangle}) \\ \wedge j_2 = \text{store}(j_1, \text{width}, 10) \\ \wedge j_3 = \text{store}(j_2, \text{height}, 10) \\ \wedge j_4 = \text{store}(j_3, \text{vtable}, \text{Square}) \\ \wedge j_5 = \text{store}(j_4, \text{width}, 10) \\ \wedge \text{return\_value}_1 = \\ (\text{select}(j_5, \text{width}) \times \text{select}(j_5, \text{width})) \end{array} \right]$$
$$P := [\text{return\_value}_1 = 100]$$

# Running Example (2)

```
Square (int w) : Rectangle(w,w)
```

contain the address  
of the object's bound  
methods




```
    *width; }
```

```
Shape *sqre = new Square(10);  
assert (sqre->area() == 100);
```

$$C := \left[ \begin{array}{l} j_1 = store(j_0, vtable, Rectanle) \\ \wedge j_2 = store(j_1, width, 10) \\ \wedge j_3 = store(j_2, height, 10) \\ \wedge j_4 = store(j_3, vtable, Square) \\ \wedge j_5 = store(j_4, width, 10) \\ \wedge return\_value_1 = \\ (select(j_5, width) \times select(j_5, width)) \end{array} \right]$$
$$P := [return\_value_1 = 100]$$

# Exception Handling (1)

- exceptions are unexpected situations within a C++ programs
  - access an invalid position in a vector throws an *out\_of\_range* exception
- exception handling is divided into three elements: a **try** block, a **catch** block, and a **throw** statement

```
int main (void) {  
    try {  try block  
        throw 1;  throw statement  
    }  
    catch (int) { return 1; }  
    catch (char) { return 2; }  catch block  
    return 0;  
}
```

# Exception Handling (2)

**try-catch** conversion to **goto** functions (internal flow)

```
main():  
  CATCH signed_int->1, char->2  
  THROW signed_int: 1  
  CATCH  
  GOTO 3
```

```
1: int #anon;  
   return 1;  
   GOTO 3
```

```
2: char #anon;  
   return 2;
```

```
3: return 0;  
   END_FUNCTION
```

jump when the type is *signed int*

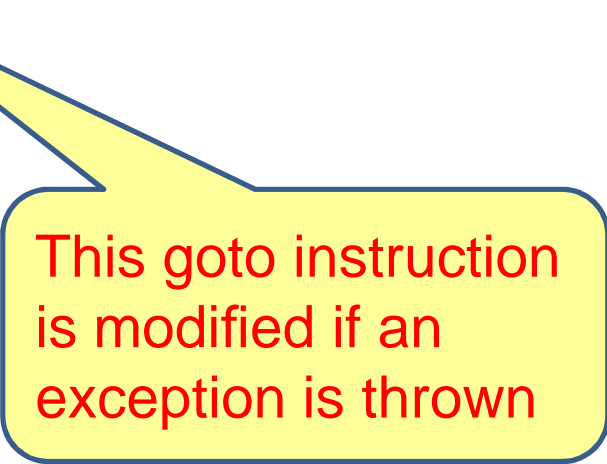
jump when the type is *char*

This goto instruction is modified if an exception is thrown

# Exception Handling (2)

**try-catch** conversion to **goto** functions (internal flow)

```
main():  
    CATCH signed_int->1, char->2  
    THROW signed_int: 1  
    CATCH  
    GOTO 1  
  
1:    int #anon;  
     return 1;  
     GOTO 3  
  
2:    char #anon;  
     return 2;  
  
3:    return 0;  
     END_FUNCTION
```



This goto instruction  
is modified if an  
exception is thrown

# Experimental Results

- Goal: compare the efficiency of C++ verification on 1165 C++ programs using ESBMC and LLBMC
- Setup:
  - ESBMC v1.20 with SMT Solver Z3 3.2
  - LLBMC 2012.2a
  - Intel Core i7-2600, 3.40 GHz with 24 GB of RAM running Ubuntu 64-bits

# About the benchmarks

		N	L	Time	P	N	FP	FN	FAIL	TO	MO
		130	3376	996	63	38	16	13	0	0	0
2	Deque	43	1239	238	19	20	0	0	0	0	0
2	Vect	14	6853	274	95	37	3	1	0	0	0
		670	2292	928	25	25	3	7	0	0	0
5	Queue	14	328	177	7	7	0	0	0	0	0
		12	28	82	6	6	0	0	0	0	0
		51	10	311	17	17	0	0	0	0	0
8	My caten	67	743	45	41	41	7	1	0	0	0
9	Stream	3	3	3	3	3	0	0	0	0	0
10	String	2	2	2	2	2	0	0	0	0	0
11	Cpp	4	4	4	4	4	4	0	0	0	0
		<b>1165</b>	<b>55953</b>	<b>58386</b>	<b>680</b>	<b>354</b>	<b>42</b>	<b>92</b>	<b>7</b>	<b>0</b>	<b>0</b>

Number of programs

Lines of code

Verification time of the modules (s)

Positive verification  
**GOOD THING**

Negative verification  
**GOOD THING**

Negative verification  
**BAD THING**

Time out  
**BAD THING**

Memory out  
**BAD THING**

Crash  
**BAD THING**

Negative verification  
**BAD THING**



# Experimental Results with ESBMC

	Testsuite	N	L	Time	P	N	FP	FN	FAIL	TO	MO
1	Algorithm	130	3376	996	63	38	16	12	0	0	0
2	Deque	43	1239	238	19	20	0	0	0	0	0
3	Vector	146	6853	2714	95	27	3	11	0	0	0
4	List	670	2292	3928	25	25	3	17	0	0	0
5	Queue	14	328	177	7	7	0	0	0	0	0
6	Stack	12	286	82	6	6	0	0	0	0	0
7	Inheritance	51	3460	311	28	17	1	2	3	0	0
8	Try catch	67	4743	45	17	41	7	2	0	0	0
9	Stream	66	1831	1892	51	13	0	2	0	0	0
10	String	233	4921	48186	100	112	5	16	0	0	0
11	Cpp	343	26624	1817	269	38	7	25	4	0	0
		<b>1165</b>	<b>55953</b>	<b>58386</b>	<b>680</b>	<b>354</b>	<b>42</b>	<b>92</b>	<b>7</b>	<b>0</b>	<b>0</b>

*STL modules*

# Experimental Results with ESBMC

	Testsuite	N	L	Time	P	N	FP	FN	FAIL	TO	MO
1	Algorithm	130	3376	996	63	38	16	13	0	0	0
2	Deque	43	1239	238	19	20	0	4	0	0	0
3	Vector	146	6853	2714	95	37	3	11	0	0	0
4	List	670	2292	3928	25	25	3	17	0	0	0
5	Queue	14	328	177	7	7				0	0
6	Stack	12	286	82	6	6				0	0
<b>7</b>	<b>Inheritance</b>	<b>51</b>	<b>3460</b>	<b>311</b>	<b>28</b>	<b>17</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>0</b>	<b>0</b>
<b>8</b>	<b>Try catch</b>	<b>67</b>	<b>4743</b>	<b>45</b>	<b>17</b>	<b>41</b>	<b>7</b>	<b>2</b>	<b>0</b>	<b>0</b>	<b>0</b>
9	Stream	66	1831	1892	51	13	0	2	0	0	0
10	String	233	4921	48186	100	112	5	16	0	0	0
11	Cpp	343	26624	1817	269	38	7	25	4	0	0
		<b>1165</b>	<b>55953</b>	<b>58386</b>	<b>680</b>	<b>354</b>	<b>42</b>	<b>92</b>	<b>7</b>	<b>0</b>	<b>0</b>

*Inheritance and exception handling*

# Experimental Results with ESBMC

	Testsuite	N	L	Time	P	N	FP	FN	FAIL	TO	MO
1	Algorithm	130	3376	996	63	38	16	13	0	0	0
2	Deque	43	1239	238	19	20	0	4	0	0	0
3	Vector	146	6853	2714	95	37	3	11	0	0	0
4	List	670	2292	3928	25	25	3	17	0	0	0
5	Queue	14	328	177	7	7	0	0	0	0	0
6	Stack	12	286	82	6	6	0	0	0	0	0
7	Inheritance	51	3460	311					3	0	0
8	Try catch	67	4743	45					0	0	0
<b>9</b>	<b>Stream</b>	<b>66</b>	<b>1831</b>	<b>1892</b>	<b>51</b>	<b>13</b>	<b>0</b>	<b>2</b>	<b>0</b>	<b>0</b>	<b>0</b>
<b>10</b>	<b>String</b>	<b>233</b>	<b>4921</b>	<b>48186</b>	<b>100</b>	<b>112</b>	<b>5</b>	<b>16</b>	<b>0</b>	<b>0</b>	<b>0</b>
11	Cpp	343	26624	1817	269	38	7	25	4	0	0
		<b>1165</b>	<b>55953</b>	<b>58386</b>	<b>680</b>	<b>354</b>	<b>42</b>	<b>92</b>	<b>7</b>	<b>0</b>	<b>0</b>

*I/O Streams  
Strings*

# Experimental Results with ESBMC

	Testsuite	N	L	Time	P	N	FP	FN	FAIL	TO	MO
1	Algorithm	130	3376	996	63	38	16	13	0	0	0
2	Deque	43	1239	238	19	20	0	4	0	0	0
3	Vector	146	6853	2714	95	37	3	11	0	0	0
4	List	670	2292	3928	25	25	3	17	0	0	0
5	Queue	14	328	177	7	7	0	0	0	0	0
6	Stack	12	286	82	6	6	0	0	0	0	0
7	Inheritance	51	3460	311	28	17	1	2	3	0	0
8	Try catch	67	4743	45	17	41	7	2	0	0	0
9	Stream	66	1831	100	11	11	0	2	0	0	0
10	String	233	4921	100	11	11	5	16	0	0	0
<b>11</b>	<b>Cpp</b>	<b>343</b>	<b>26624</b>	<b>1817</b>	<b>269</b>	<b>38</b>	<b>7</b>	<b>25</b>	<b>4</b>	<b>0</b>	<b>0</b>
		<b>1165</b>	<b>55953</b>	<b>58386</b>	<b>680</b>	<b>354</b>	<b>42</b>	<b>92</b>	<b>7</b>	<b>0</b>	<b>0</b>

*Generic programs from Deitel*

# Comparison between ESBMC and LLBMC

	Testsuite	Time	P	N	FP	FN	FAIL	TO	MO
1	Algorithm	996	63	38	16	13	0	0	0
2	Deque	238	19	20	0	4	0	0	0
3	Vector	2714	95	37	3	11	0	0	0
4	List	3928	25	25	3	17	0	0	0
5	Queue	177	7	7	0	0	0	0	0
6	Stack	82	6	6	0	0	0	0	0
		<b>8135</b>	<b>215</b>	<b>133</b>	<b>22</b>	<b>45</b>	<b>0</b>	<b>0</b>	<b>0</b>
1	Algorithm	22964	53	45	1	5	0	24	2
2	Deque	8585	16	17	0	0	1	9	0
3	Vector	7234	91	38	1	3	4	6	3
4	List	2562	5	26	5	30	0	0	4
5	Queue	45	6	7	0	1	0	0	0
6	Stack	45	6	6	0	0	0	0	0
		<b>41435</b>	<b>177</b>	<b>139</b>	<b>7</b>	<b>39</b>	<b>5</b>	<b>39</b>	<b>9</b>

ESBMC

LLBMC

# Comparison between ESBMC and LLBMC

	Testsuite	Time	P	N	FP	FN	FAIL	TO	MO
1	Inheritance	311	28	17	1	2	3	0	0
2	Try catch	45	17	41	7	2	0	0	0
		<b>356</b>	<b>45</b>	<b>58</b>	<b>8</b>	<b>4</b>	<b>3</b>	<b>0</b>	<b>0</b>
1	Inheritance	122	32	12	1	3	3	0	0
2	Try catch	4	0	1	0	0	66	0	0
		<b>126</b>	<b>32</b>	<b>13</b>	<b>1</b>	<b>3</b>	<b>69</b>	<b>0</b>	<b>0</b>

ESBMC

LLBMC

# Comparison between ESBMC and LLBMC

	Testsuite	Time	P	N	FP	FN	FAIL	TO	MO
1	Stream	1892	51	13	0	2	0	0	0
2	String	46186	100	112	5	16	0	0	0
		<b>48078</b>	<b>151</b>	<b>125</b>	<b>5</b>	<b>18</b>	<b>0</b>	<b>0</b>	<b>0</b>
1	Stream	11	17	13	0	35	1	0	0
2	String	37	6	121	4	102	0	0	0
		<b>48</b>	<b>23</b>	<b>134</b>	<b>4</b>	<b>137</b>	<b>1</b>	<b>0</b>	<b>0</b>

ESBMC

LLBMC

# Comparison between ESBMC and LLBMC

	Testsuite	Time	P	N	FP	FN	FAIL	TO	MO
1	Cpp	1817	269	38	7	25	4	0	0
-----		<b>58386</b>	<b>680</b>	<b>354</b>	<b>42</b>	<b>92</b>	<b>7</b>	<b>0</b>	<b>0</b>
1	Cpp	3260	235	24	10	56	15	2	1
		<b>44869</b>	<b>467</b>	<b>310</b>	<b>22</b>	<b>235</b>	<b>90</b>	<b>41</b>	<b>10</b>

The table is annotated with two yellow callout boxes on the right side. The top callout, labeled 'ESBMC', is connected by a bracket to the first row of the table. The bottom callout, labeled 'LLBMC', is connected by a bracket to the last row of the table. The summary row (row 3) is separated from the other rows by a dashed line.

- ESBMC++ took approximately 16 hours and successfully verified 1046 out of 1165 (**89%**)
- LLBMC took approximately 12 hours and successfully verified 777 out of 1165 (**66%**)



# Experimental Results Sniffer Code

- ESBMC++ was used to verify a **commercial** application provided by Nokia Institute of Technology (INdT)
- The sniffer code contains 20 classes, 85 methods, and approximately 2839 lines of C++ code
- Five bugs were identified that were related to arithmetic under- and over-flow. The bugs were later confirmed by the developers

# Conclusions

- SMT-based verification of C++ programs by focusing on the major features of the language
- Described the implementation of STL containers, inheritance, polymorphism and exception handling
  - in particular, exception specification, which is a feature that is not supported by others BMC tools
- ESBMC++ outperforms LLBMC if we consider the verification of C++ programs
  - with increased accuracy (i.e. exception enabled verification)
- Also, ESBMC++ was able to find undiscovered bugs in the sniffer code, a commercial application