Using LLVM in a Model Checking Workflow

Gyula Sallai

2018 European LLVM Developers Meeting

MŰEGYETEM 1782

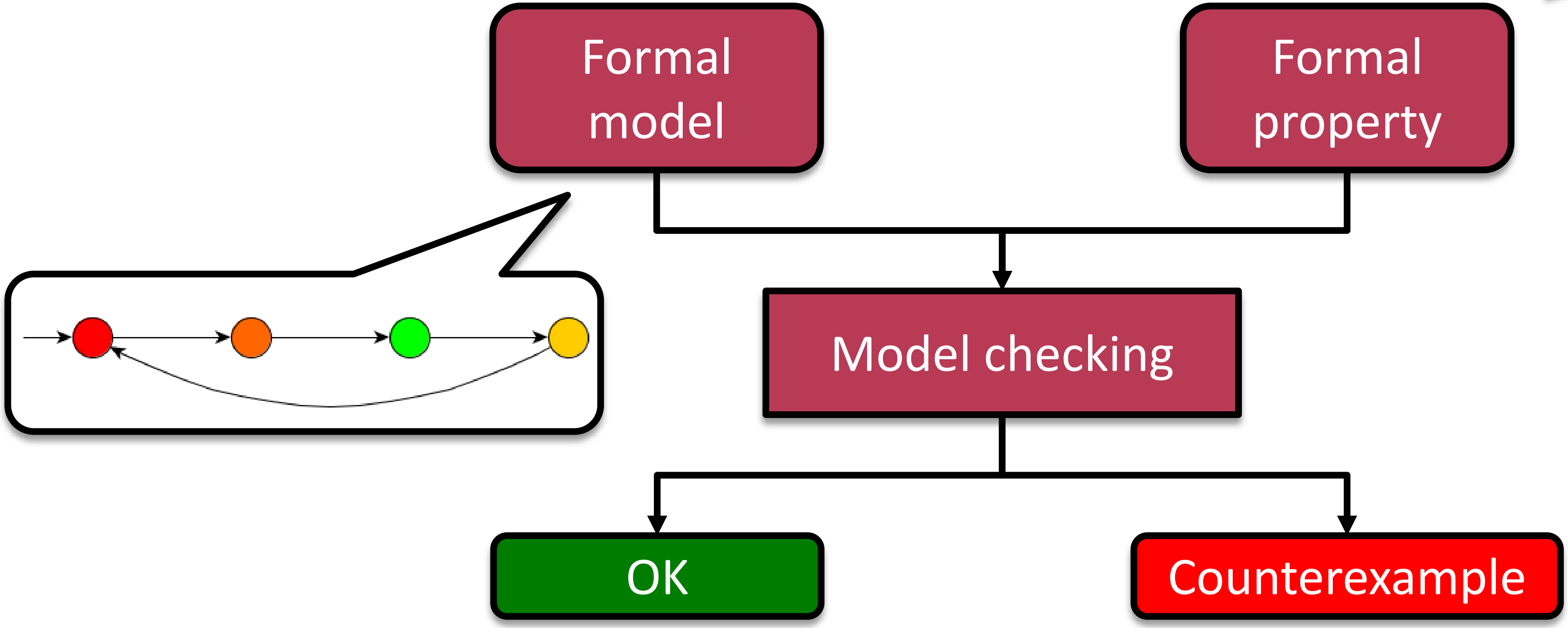
EMBERI ERŐFORRÁSOK
MINISZTÉRIUMA

Introduction

Motivation

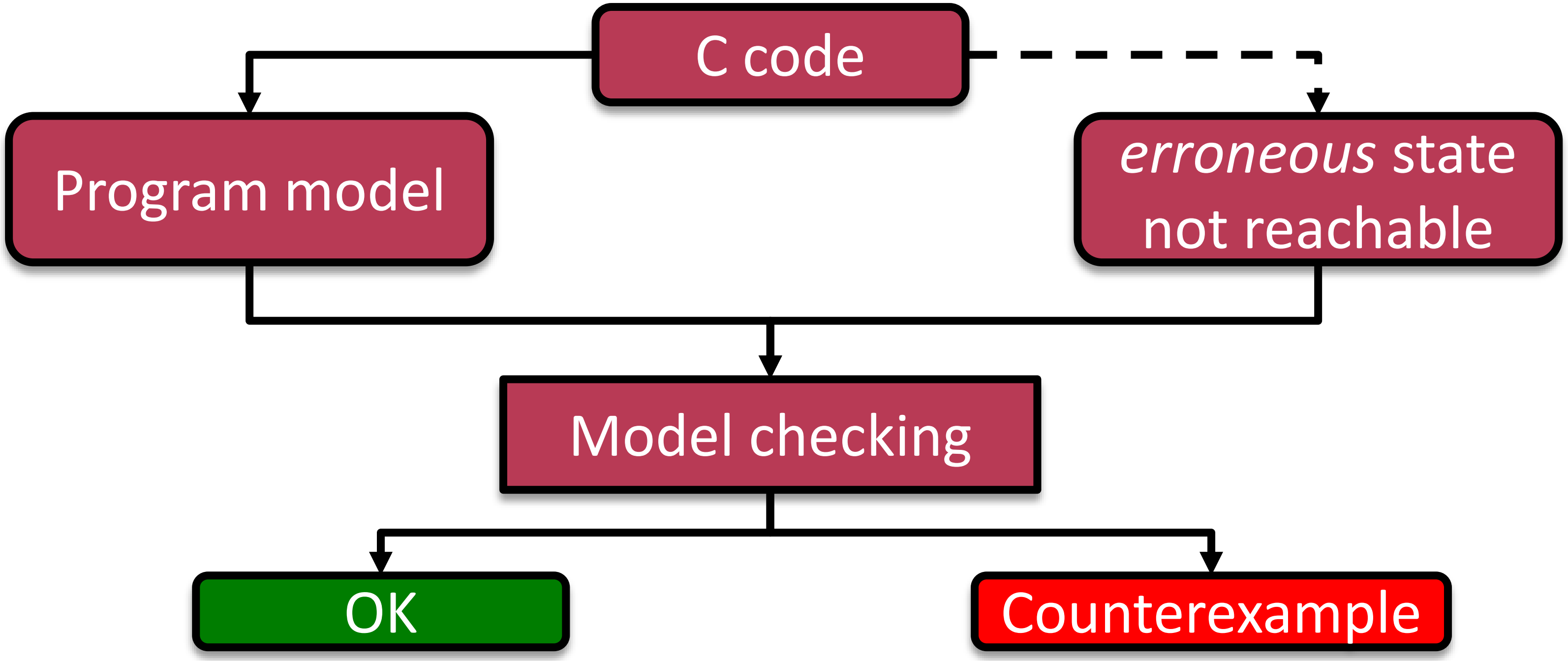
- Embedded software systems
  - Usually written in C
- Confidence in correctness?

$\neg(\text{Red} \wedge \text{Green})$



Software model checking

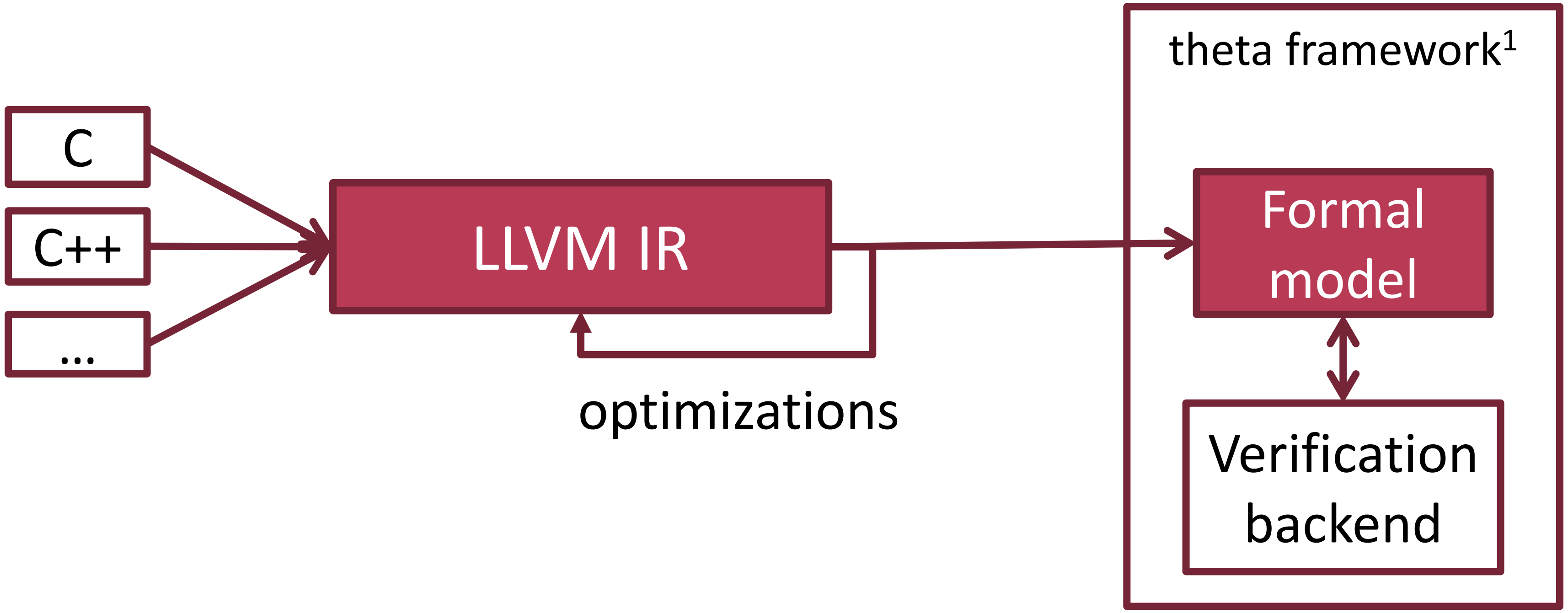
- Automatic transformation from source code



- Model checking is computationally hard
  - Undecidable in general
  - Model size/complexity must be reduced

LLVM for model checking

- LLVM IR as a language frontend?
  - Language-agnostic
  - Optimization infrastructure
- Using LLVM IR for model checking



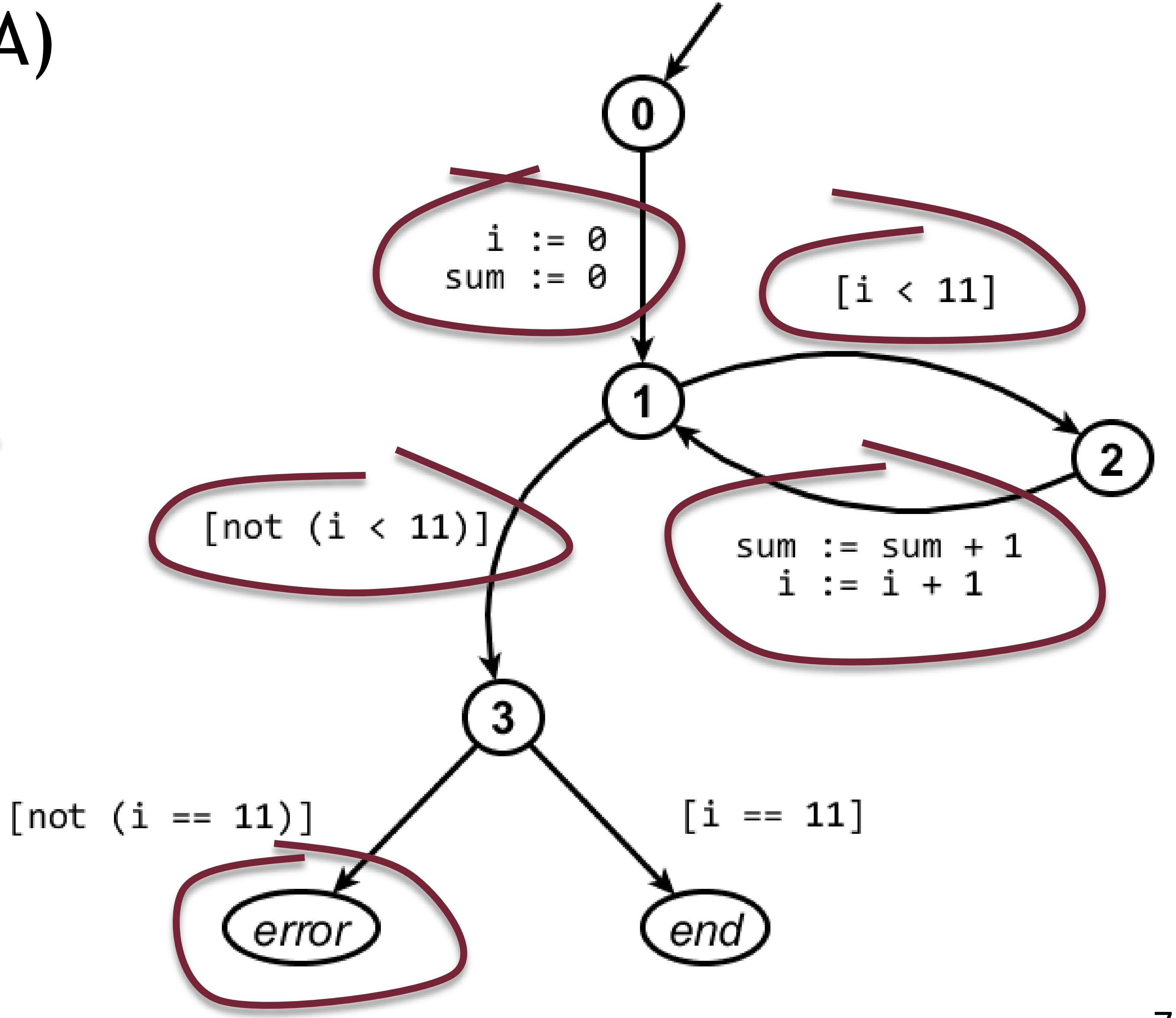
[1] https://github.com/ftsrg/theta

Transformation to formal models

Formal model for computer programs

- Control flow automata (CFA)

```
int i = 0;
int sum = 0;
while (i < 11) {
    sum = sum + i;
    i = i + 1;
}
assert(i == 11);
```



- *error*: failing assertions

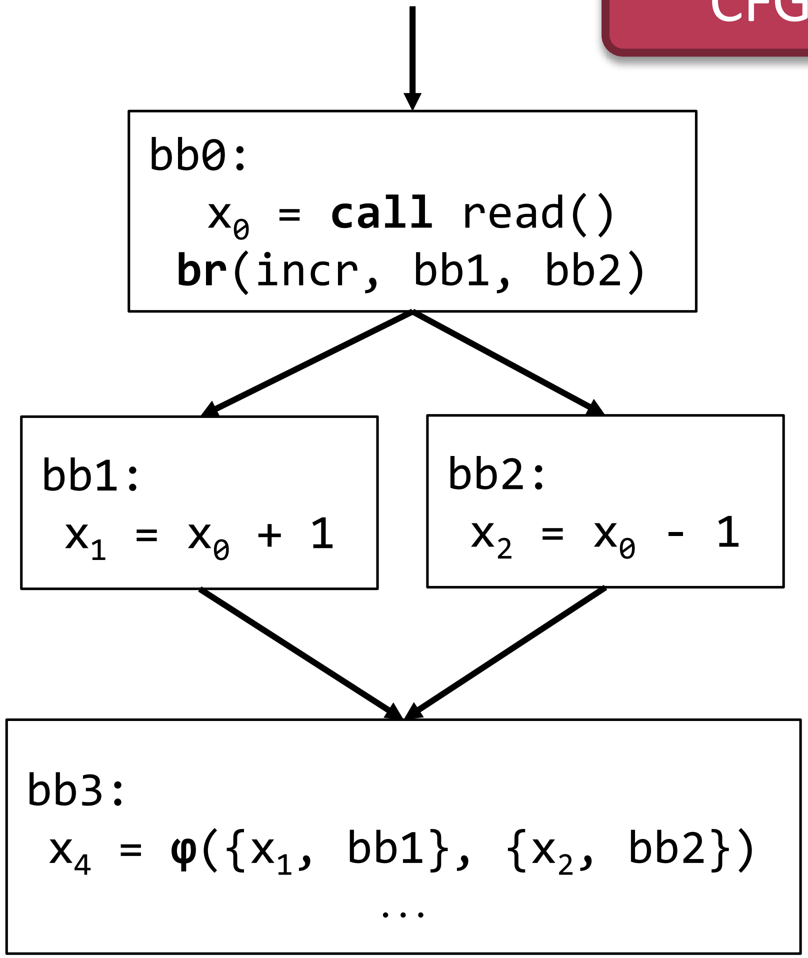
LLVM IR to formal models

- Gap between the IR and formal models
  - Designed for compilation ⇔ designed for theorem provers

- LLVM IR has more expressive power
  - SSA, φ-nodes → transformation rules
  - Pointers → theory of arrays, integer addresses
  - Global variables → promotion to locals
  - Procedure calls → function inlining

LLVM IR to formal models

CFG



bb0:

x_0 = **call** read()

br(incr, bb1, bb2)

bb1:

$x_1 = x_0 + 1$

bb2:

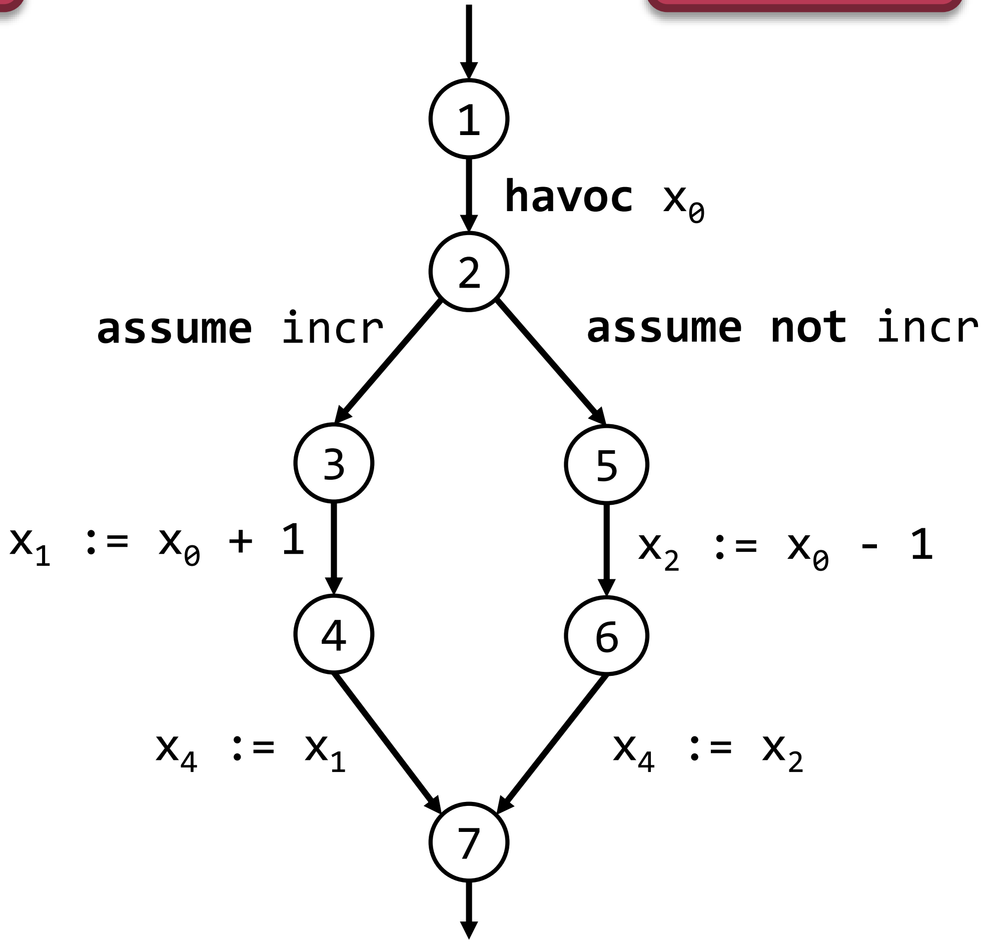
$x_2 = x_0 - 1$

bb3:

$x_4 = \varphi(\{x_1, \text{bb1}\}, \{x_2, \text{bb2}\})$

...

CFA



havoc x_0

assume incr

assume not incr

$x_1 := x_0 + 1$

$x_2 := x_0 - 1$

$x_4 := x_1$

$x_4 := x_2$

Optimization algorithms

Optimizations

- Need to be configurable

- Optimizations in LLVM
  - Constant propagation, dead code elimination
  - Function inlining

- Other transformations
  - Global variables to locals
  - Program slicing

Program slicing

- **Slice:** subprogram, which produces the same output and assigns the same values to a set of variables as the original program.

```
0: int i = 0;
1: int x = 0;
2: while (i < 11) {
3:      x = x + i;
4:      i = i + 1;
   }
5:  assert(i != 0);
```

```
0: int i = 0;
~~1: int x = 0;~~
2: while (i < 11) {
~~3:      x = x + i;~~
4:      i = i + 1;
   }
5:  assert(i != 0);
```

Criterion: *value of **i** at statement 5*

Evaluation

Evaluation

- SV-Comp: Competition on Software Verification[1]
  - Verification tasks written in C

- Program categories
  - *locks*: locking mechanisms
  - *eca*: event-driven systems
  - *ssh*: ssh protocol

[1] https://sv-comp.sosy-lab.org/2016/

Evaluation

*Opt: with optimizations
*Slice: with slicing

Model	Vars	Locs	VarsOpt	LocsOpt	#Slice	VarSlice	LocsSlice
locks10	55	236	52	231	10	**5.5**	**27**
locks14	75	324	72	319	14	**5.5**	**26.5**
eca1	1104	2937	976	2870	1	614	1908
eca2	1040	2854	892	2778	1	590	1936
eca3	3269	10719	2781	10325	1	2408	9050
ssh1	196	693	174	648	1	109	394

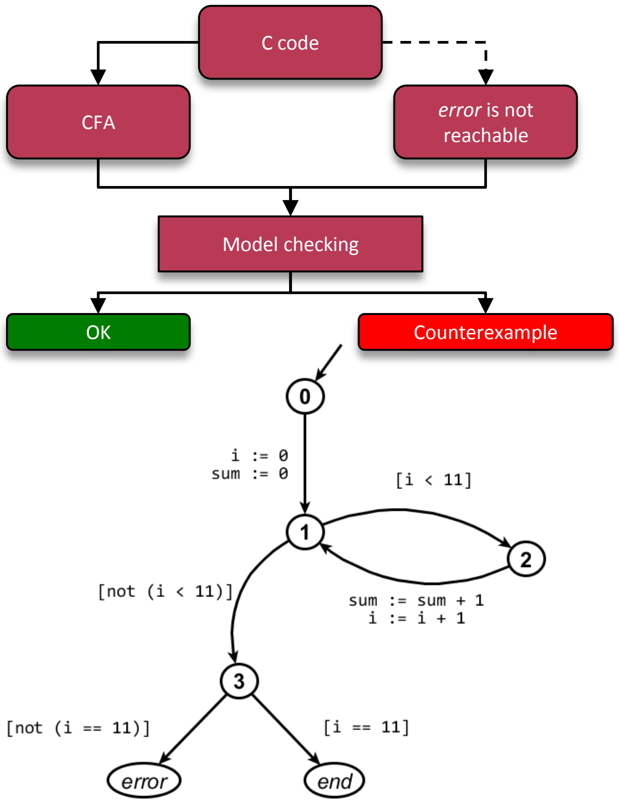
Many small slices

Some reduction with optimizations, more with slicing

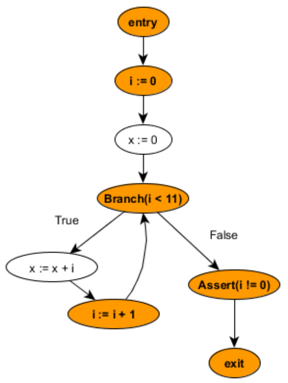
Significant reduction

Summary

- Software model checking
- LLVM IR-based model checking
  - Transformation to formal models
  - Configurable optimizations
  - Program slicing
- Future work
  - Improved pointer support
  - New slicing methods (heuristics…)



```
0: int i = 0;
~~1: int x = 0;~~
2: while (i < 11) {
~~3:        x = x + i;~~
4:        i = i + 1;
    }
5: assert(i != 0);
```


entry
i := 0
x := 0
Branch(i < 11)
True
False
x := x + i
i := i + 1
Assert(i != 0)
exit