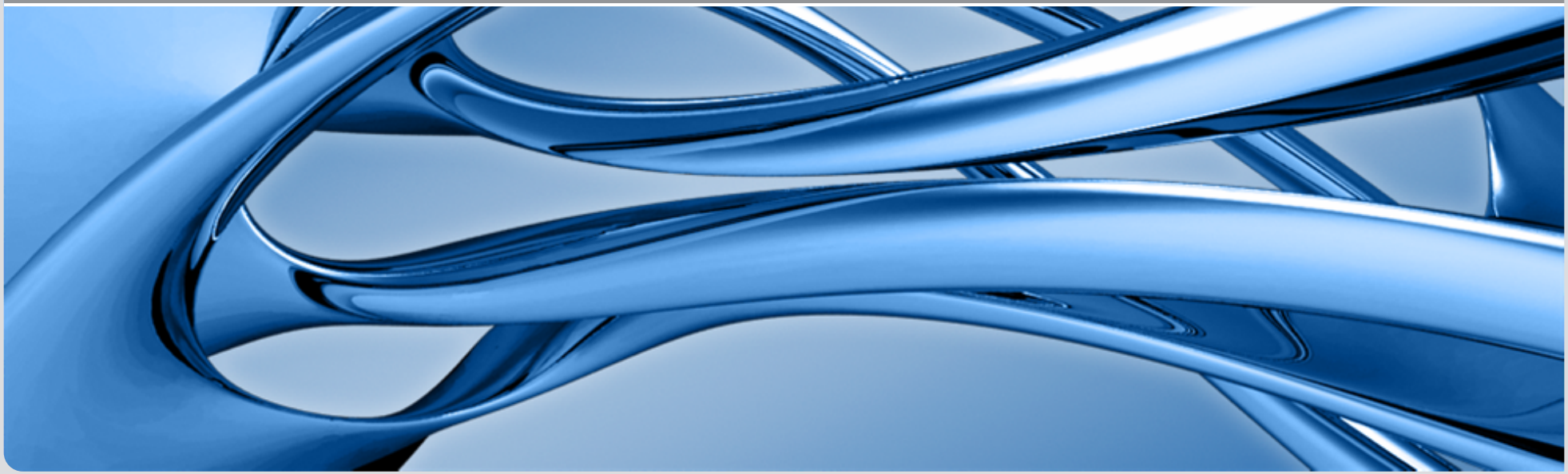
Karlsruhe Institute of Technology

# LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR

***Florian Merz*, Stephan Falke, Carsten Sinz**

INSTITUTE FOR THEORETICAL COMPUTER SCIENCE

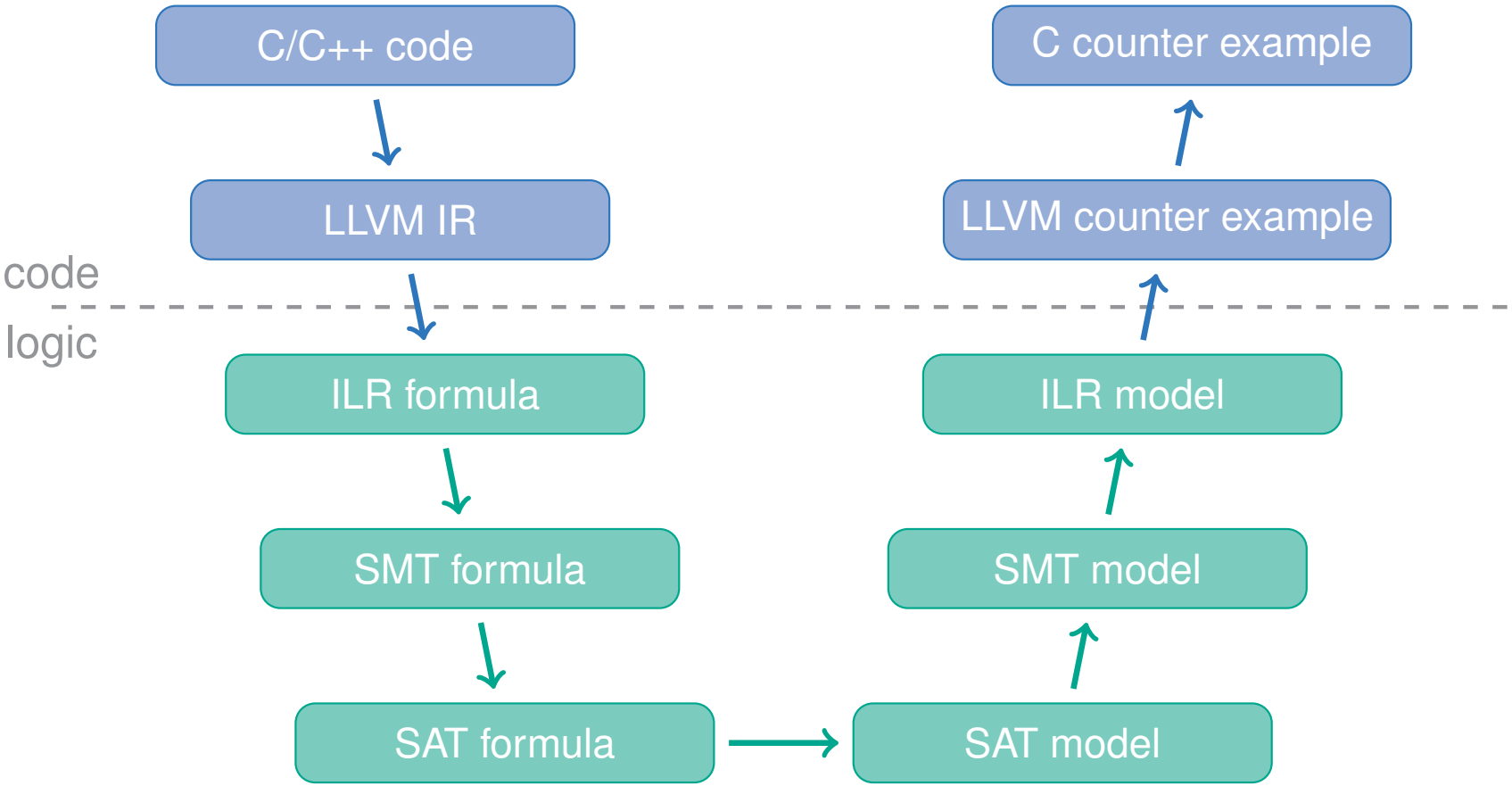


# LLBMC – Low-level Software Bounded Model Checking

Karlsruhe Institute of Technology

- Method: Software bounded model checking
- Target: (embedded) C/C++ code
- Key feature: Bit-precision
- Built-in checks:
  - Div-by-zero
  - Arithmetic overflow
  - Shift overflow
  - Memory access correctness
  - Memory allocation correctness
  - ...

# Software Bounded Model Checking in LLBMC



# From Code to Logic

Goal: Show that

$$C \rightarrow P$$

where $C$ is the program's source code, and $P$ are the program's desired properties.

# From Code to Logic

Goal: Show that

$$C \rightarrow P$$

To do:

Encode the desired properties $P$ ☐

Encode the program's source code $C$ ☐

# Encoding Properties

E.g., for instruction I

```
I: int z = x / y;
```

we need to check

$$y \neq 0.$$

# Encoding Properties

E.g., for instruction I

```
I: int z = x / y;
```

we need to check

$$c_{exec}(I) \rightarrow y \neq 0.$$

$$c_{exec}(I) = \begin{cases} \text{true} & \text{if instruction } I \text{ is executed} \\ \text{false} & \text{otherwise} \end{cases}$$

# Encoding Properties

E.g., for instruction I

```
I: int z = x / y;
```

we need to check

$$c_{exec}(I) \rightarrow y \neq 0.$$

$$c_{exec}(I) = \begin{cases} \text{true} & \text{if instruction } I \text{ is executed} \\ \text{false} & \text{otherwise} \end{cases}$$

$c_{exec}$ guards the assertion $y \neq 0$. But where to get $c_{exec}$ from?

# From Code to Logic

Goal: Show that

$$C \rightarrow P$$

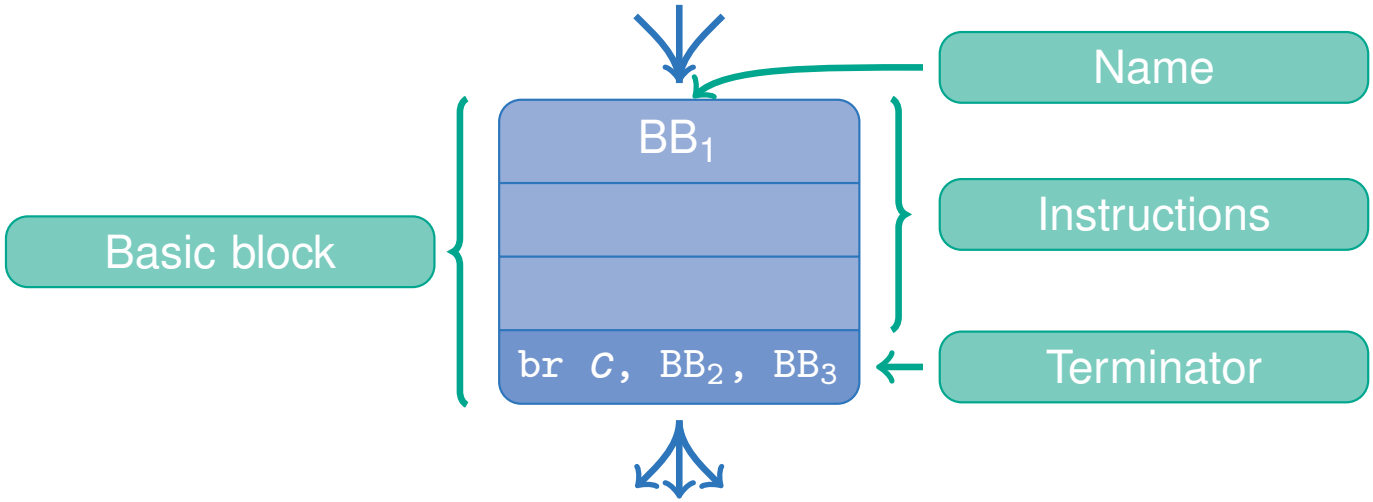
To do:

Encode the desired properties $P$ ☐

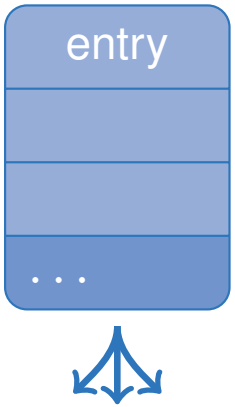
- Encoding the checks ☑
- Guarding the checks ☐

Encode the program's source code $C$ ☐

# Basic Blocks

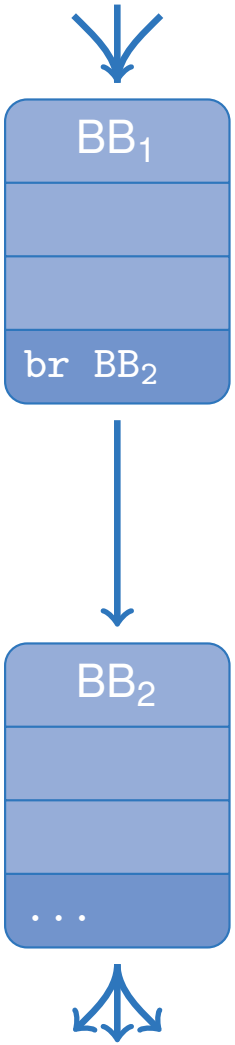


# Guards – The Function's `entry` Block



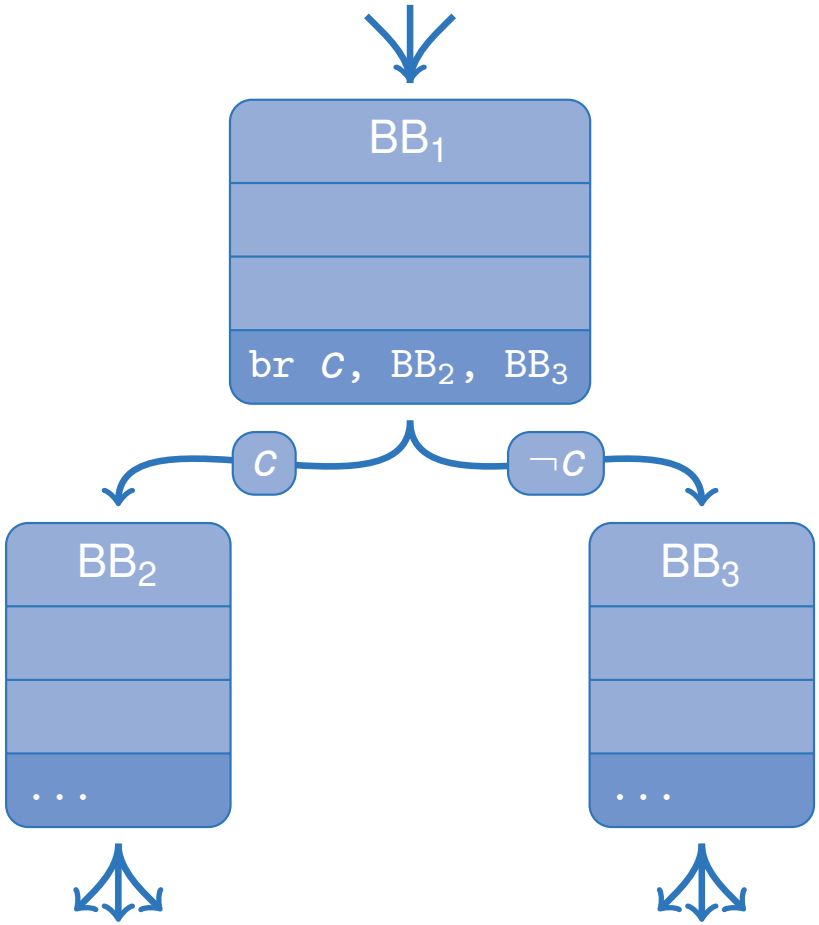

$$c_{exec}(entry) = true$$

# Guards – Unconditional Branches




$$c_{exec}(BB_2) = c_{exec}(BB_1)$$

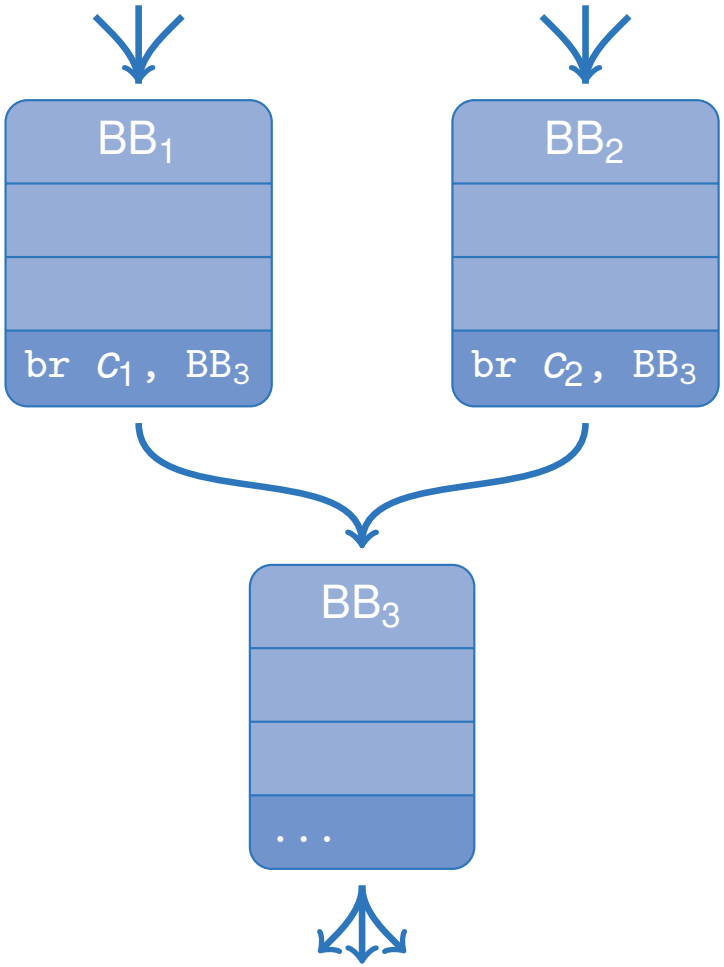
# Guards – Conditional Branches




$$c_{exec}(BB_2) = c_{exec}(BB_1) \wedge c$$

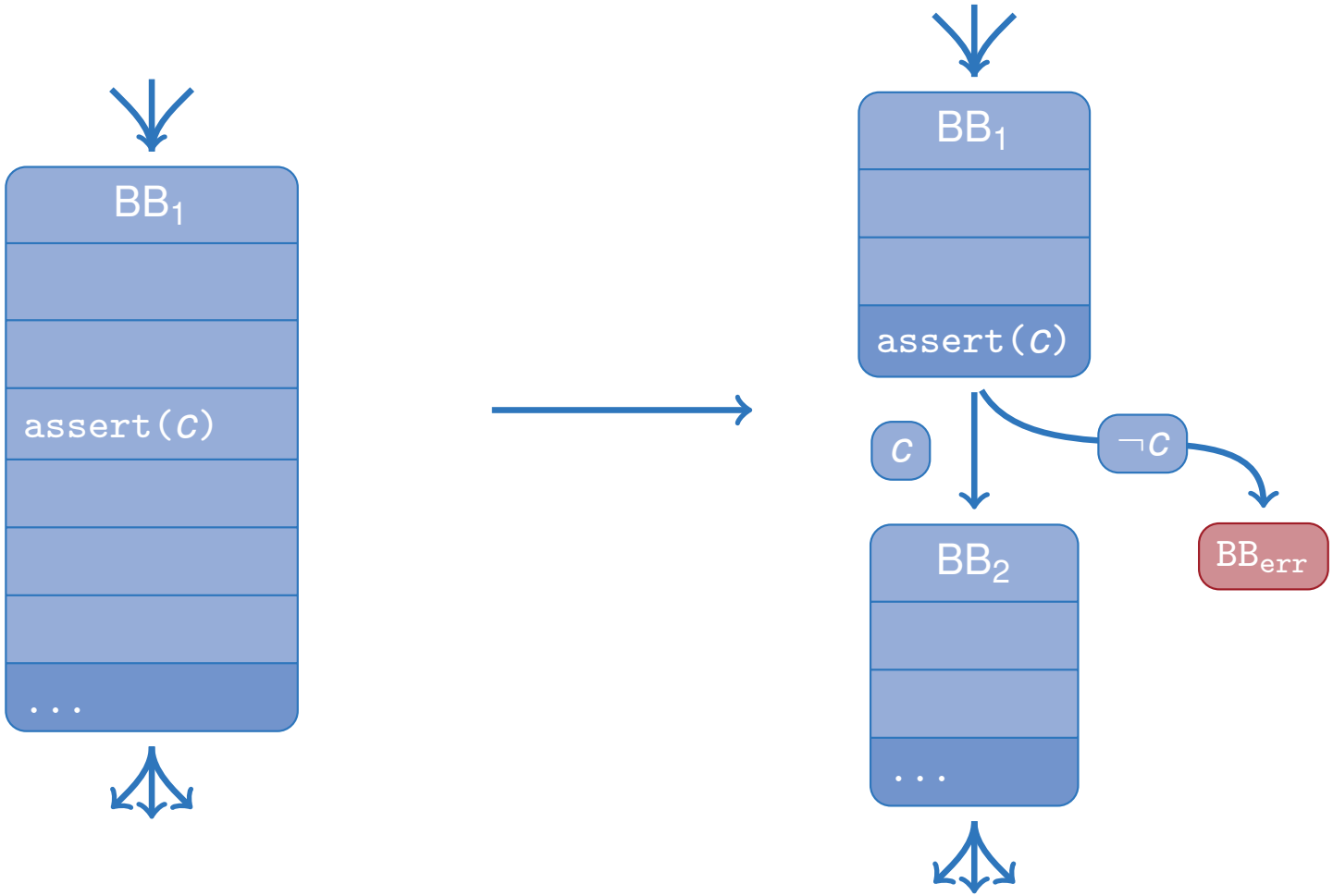
$$c_{exec}(BB_3) = c_{exec}(BB_1) \wedge \neg c$$

# Guards – Multiple Predecessors




$$c_{exec}(BB_3) = c_{exec}(BB_1) \wedge c_1 \vee c_{exec}(BB_2) \wedge c_2$$

# Guards – Assertions and Assumptions



# From Code to Logic

Goal: Show that

$$C \rightarrow P$$

To do:

Encode the desired properties $P$ ☑

- Encoding the checks ☑
- Guarding the checks ☑

Encoding the source code $C$ ☐

# From Code to Logic

Karlsruhe Institute of Technology

## Original Code:

```
x := 0;
x := 1;
y := x + 1;
```

# From Code to Logic

Karlsruhe Institute of Technology

## Original Code:

```
x := 0;
x := 1;
y := x + 1;
```

## SSA Form:

$$
\begin{aligned}
x_0 &:= 0;\\
x_1 &:= 1;\\
y_0 &:= x_1 + 1;
\end{aligned}
$$

# From Code to Logic

Karlsruhe Institute of Technology

## Original Code:

```
x := 0;
x := 1;
y := x + 1;
```

## SSA Form:

$$x_0 := 0;$$
$$x_1 := 1;$$
$$y_0 := x_1 + 1;$$

## Logic:

$$x_0 = 0$$
$$x_1 = 1$$
$$y_0 = x_1 + 1$$

# From Code to Logic

Karlsruhe Institute of Technology

## Original Code:

```
x := 0;
x := 1;
y := x + 1;
```

## SSA Form:

```
x0 := 0;
x1 := 1;
y0 := x1 + 1;
```

## Logic:

$$x_0 = 0$$
$$x_1 = 1$$
$$y_0 = x_1 + 1$$

## Observation:

In SSA form, assignments can be read as equalities

# From Code to Logic

Karlsruhe Institute of Technology

Original Code:

```
x := 0;
x := 1;
y := x + 1;
```

SSA Form:

```
x0 := 0;
x1 := 1;
y0 := x1 + 1;
```

Logic:

$$x_0 = 0$$
$$x_1 = 1$$
$$y_0 = x_1 + 1$$

Observation:

In SSA form, assignments can be read as equalities

Idea:

Convert the whole program to some kind of SSA form

# From Code to Logic

Goal: Show that

$$C \rightarrow P$$

To do:

Encode the desired properties $P$ ☑

- Encoding the checks ☑
- Guarding the checks ☑

Encoding the source code $C$ ☐

- SSA for calculations ☐
- SSA across basic block boundaries ☐
- Memory accesses ☐
- Memory allocations/deallocations ☐

# SSA For Calculations

C-Code:

```
1  int foo(int x)
2  {
3    x = x*x;
4    x = x / 2;
5    x = x + 1;
6    return x;
7  }
```

LLVM-IR:

```
1  define i32 @foo(i32 %x) {
2    %1 = mul i32 %x, %x
3    %2 = sdiv i32 %1, 2
4    %3 = add i32 %2, 1
5    ret i32 %3
6  }
```

Solution:

Let LLVM do it

# From Code to Logic

Goal: Show that

$$C \rightarrow P$$

To do:

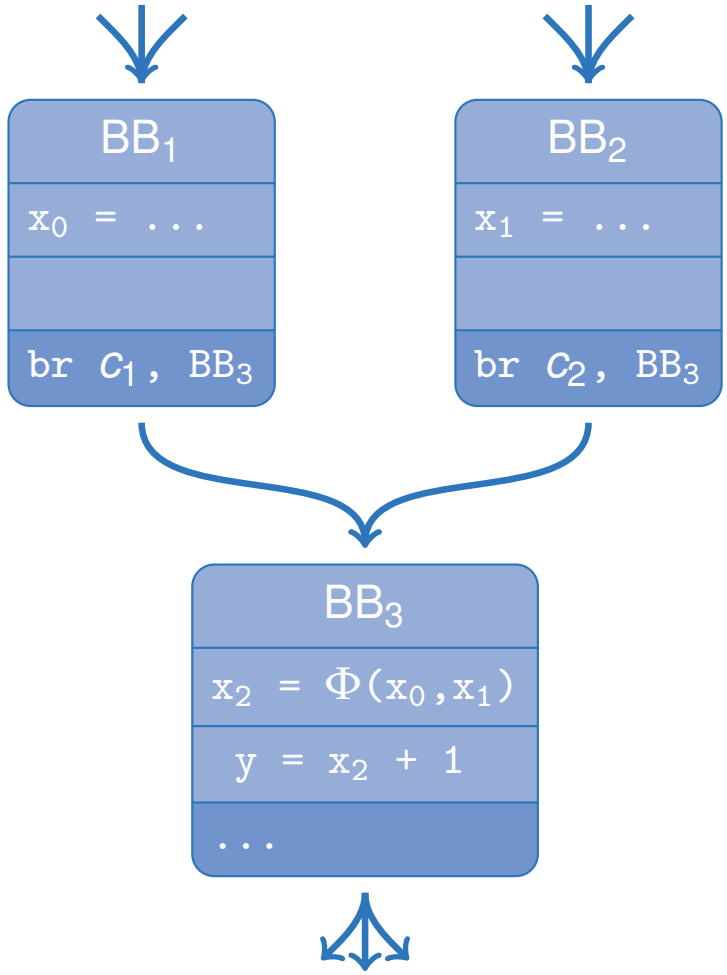
Encode the desired properties $P$ ☑

- Encoding the checks ☑
- Guarding the checks ☑

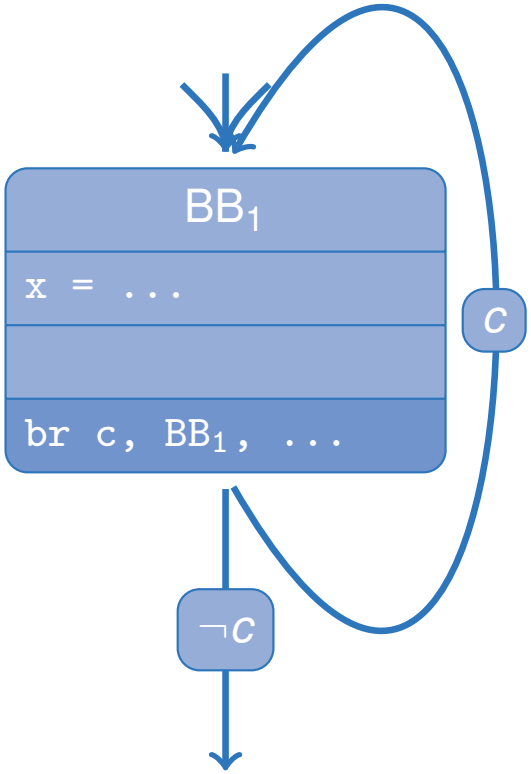
Encoding the source code $C$ ☐

- SSA for calculations ☑
- SSA across basic block boundaries ☐
- Memory accesses ☐
- Memory allocations/deallocations ☐

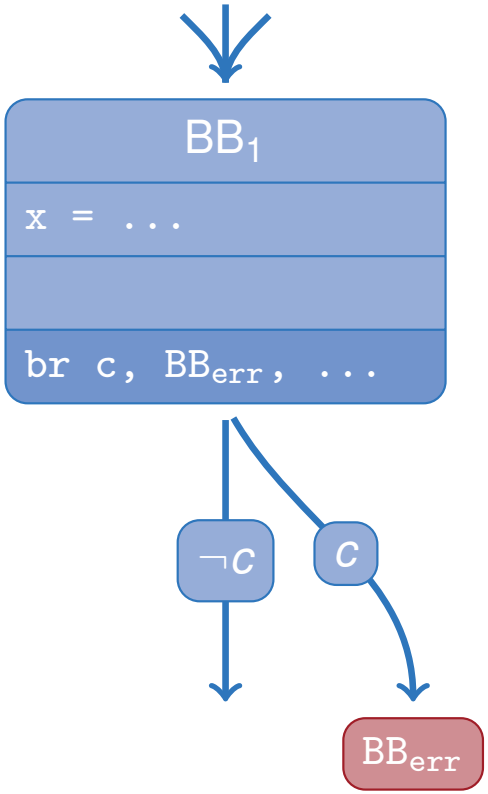
# SSA and Branches



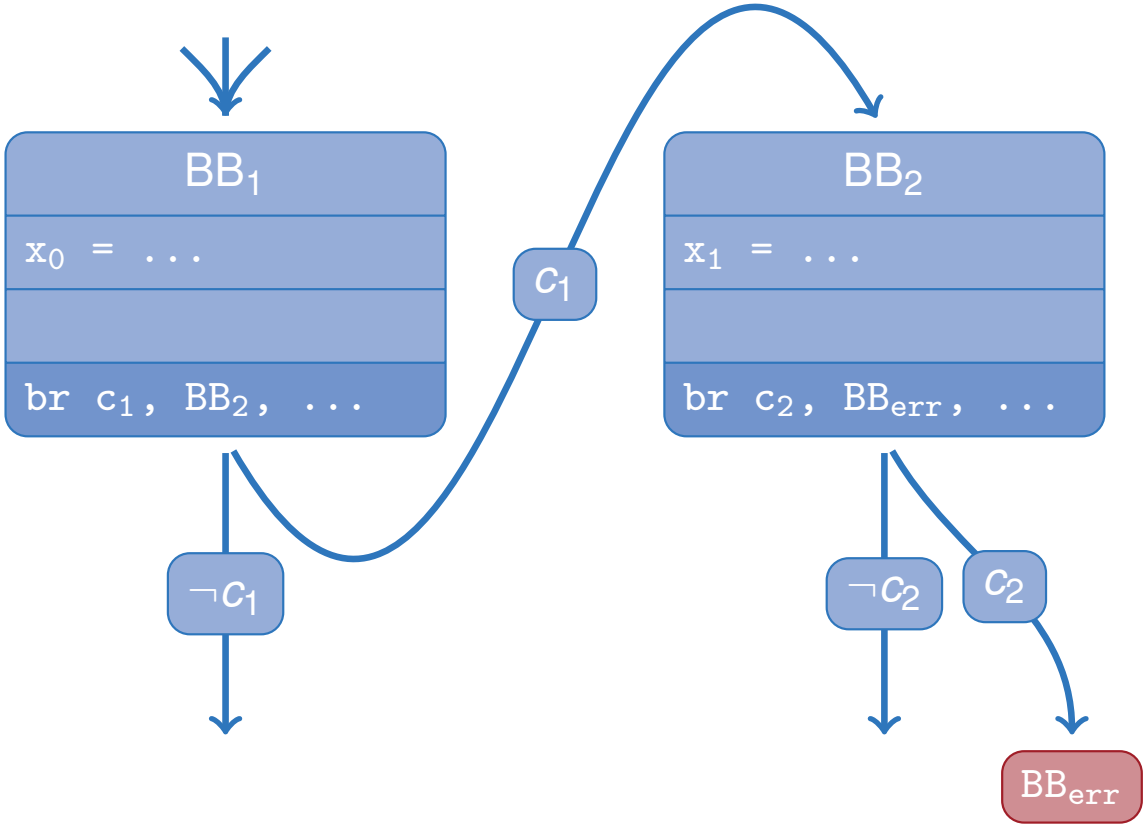
# SSA and Loops



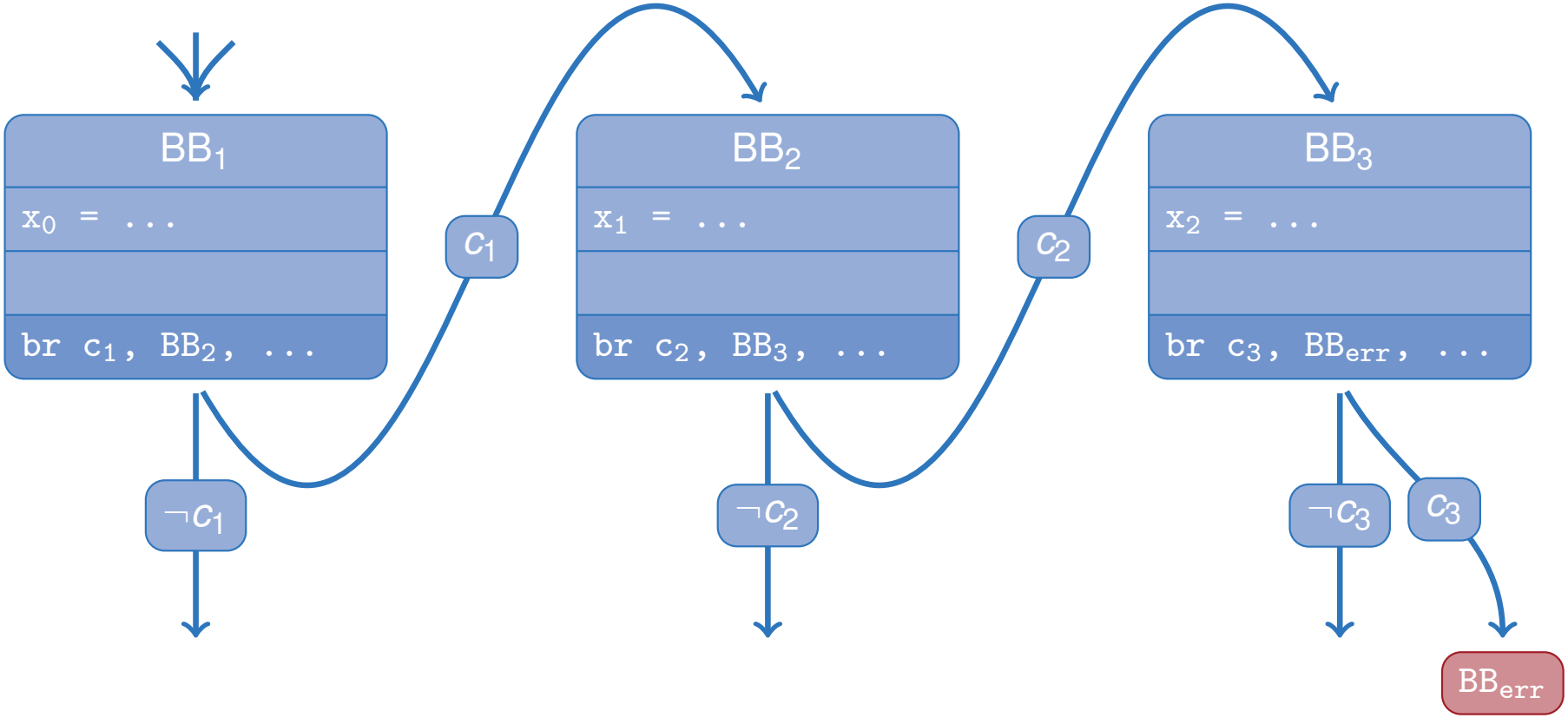
# SSA and Loops



# SSA and Loops



# SSA and Loops



# From Code to Logic

Goal: Show that

$$C \rightarrow P$$

To do:

Encode the desired properties $P$ ☑

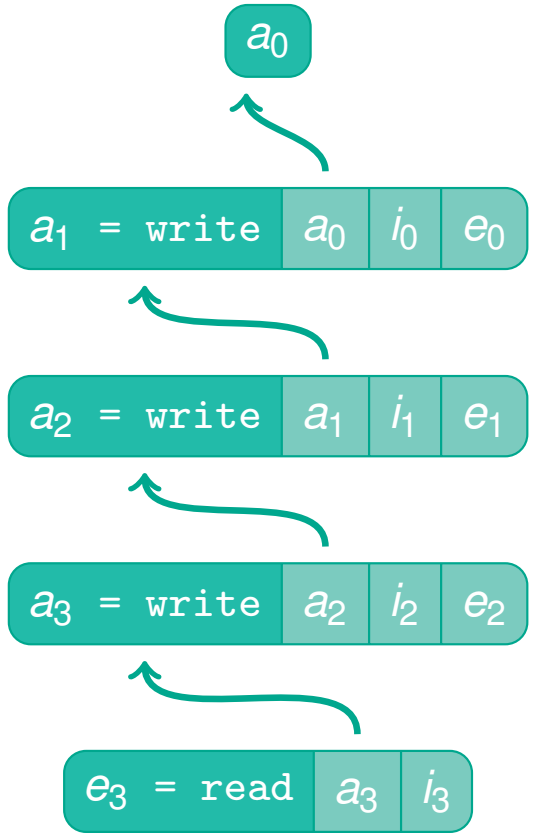
- Encoding the checks ☑
- Guarding the checks ☑

Encoding the source code $C$ ☐

- SSA for calculations ☑
- SSA across basic block boundaries ☑
- Memory accesses ☐
- Memory allocations/deallocations ☐

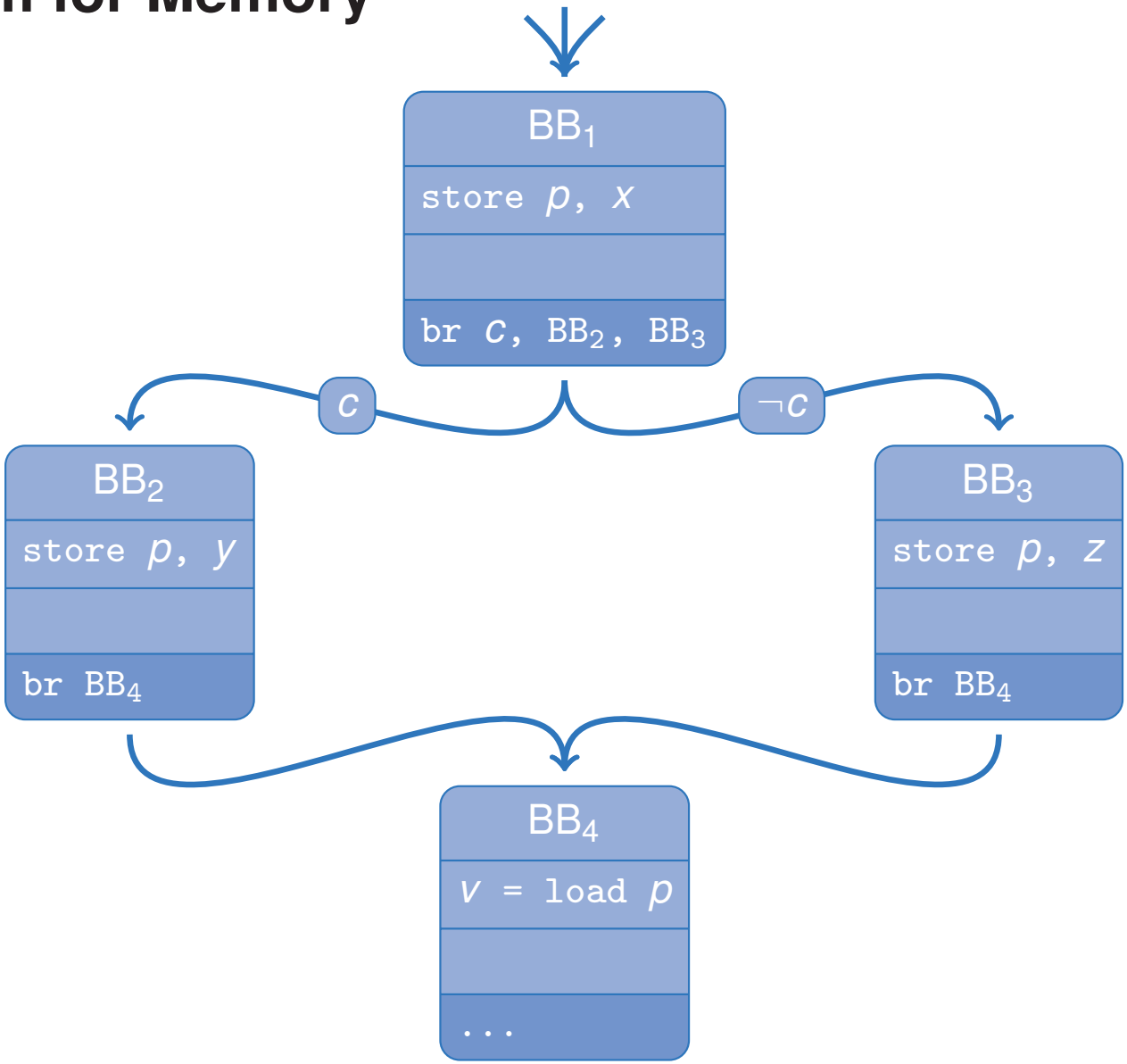
# Excursion: McCarthy's Theory of Arrays ($T_a$)

$$write : A \times I \times E \to A$$
$$read : A \times I \to E$$

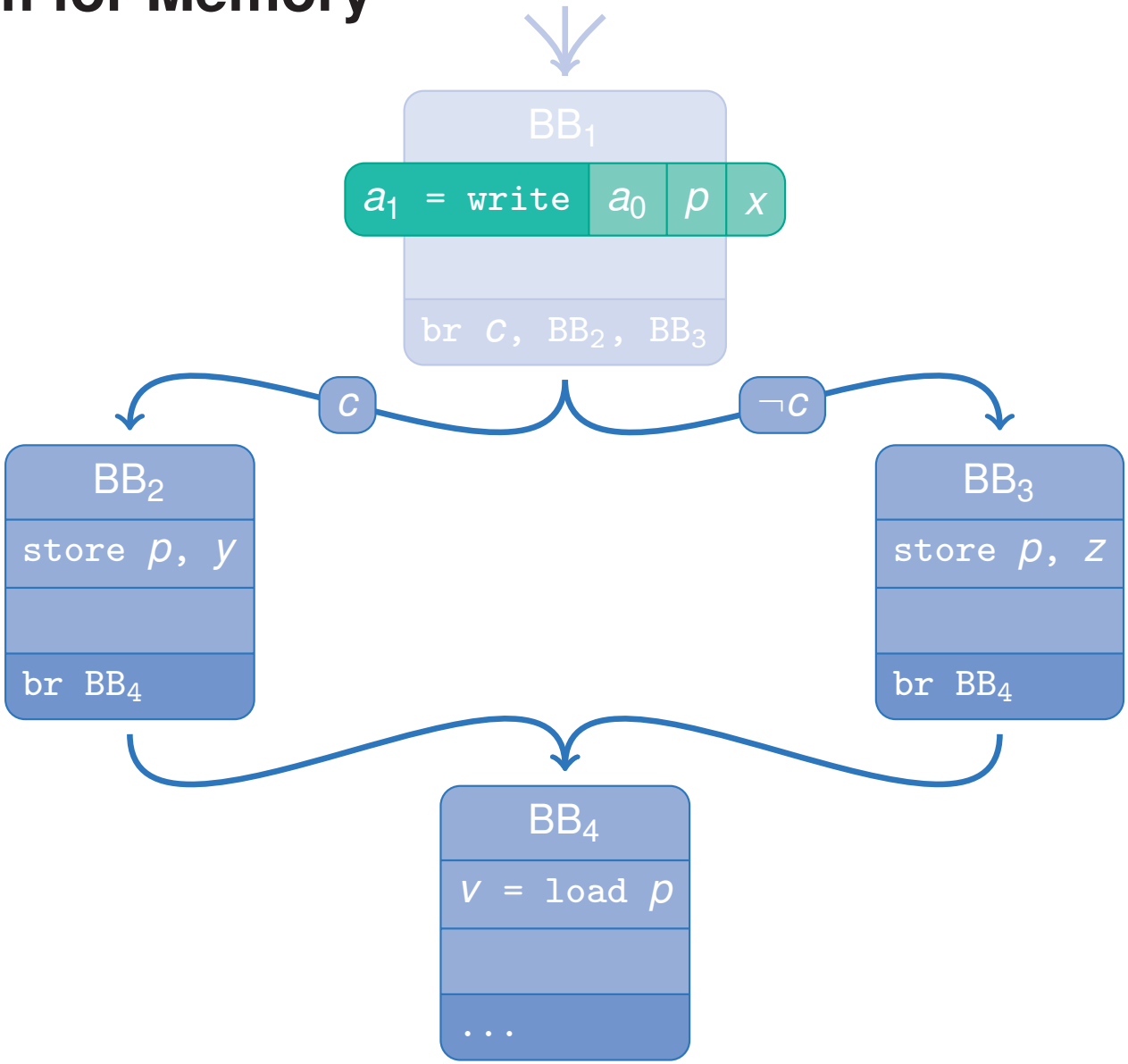



$$read(write(write(write(a_0, i_0, e_0), i_1, e_1), i_2, e_2), i_3)$$

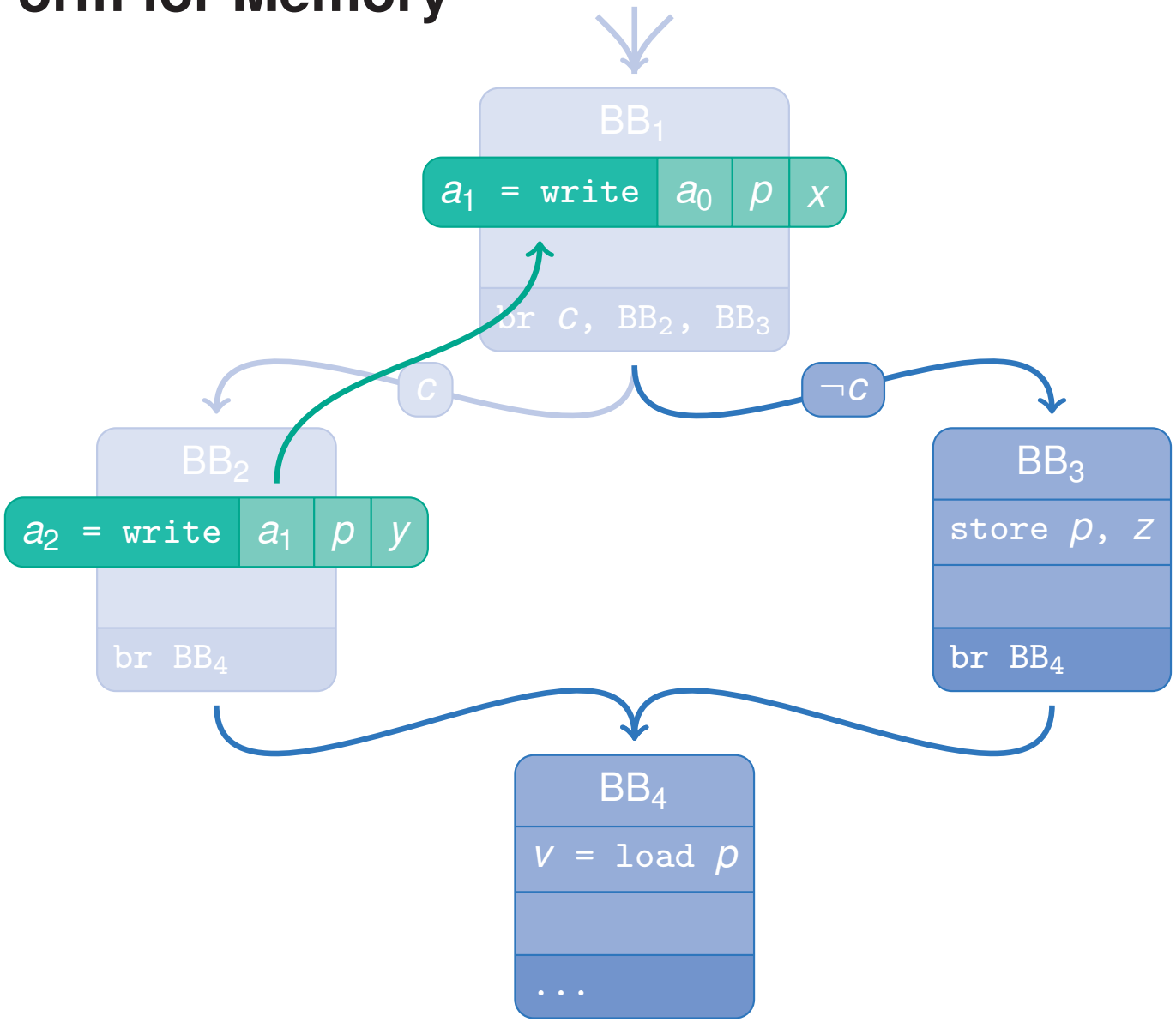
# SSA Form for Memory



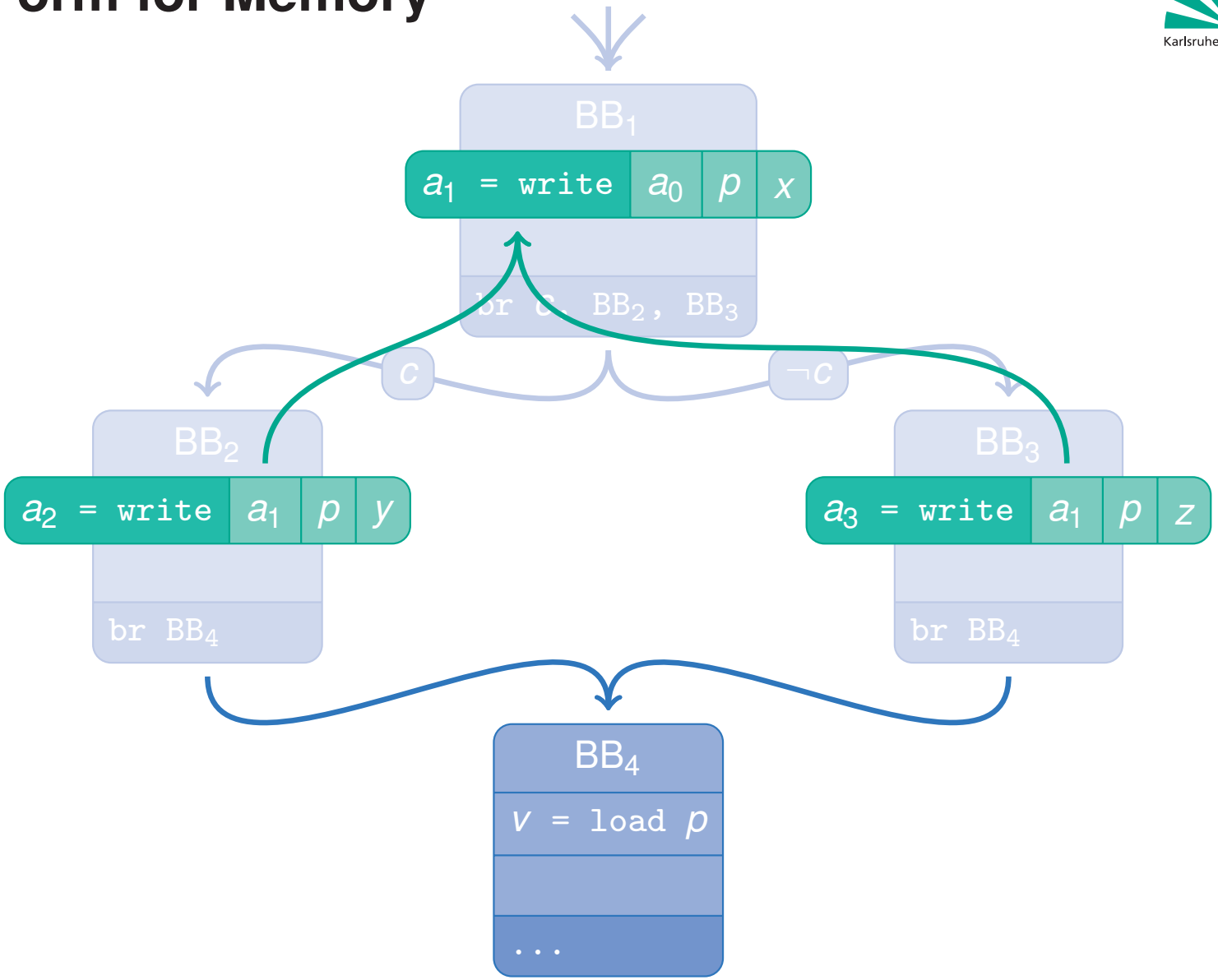
# SSA Form for Memory



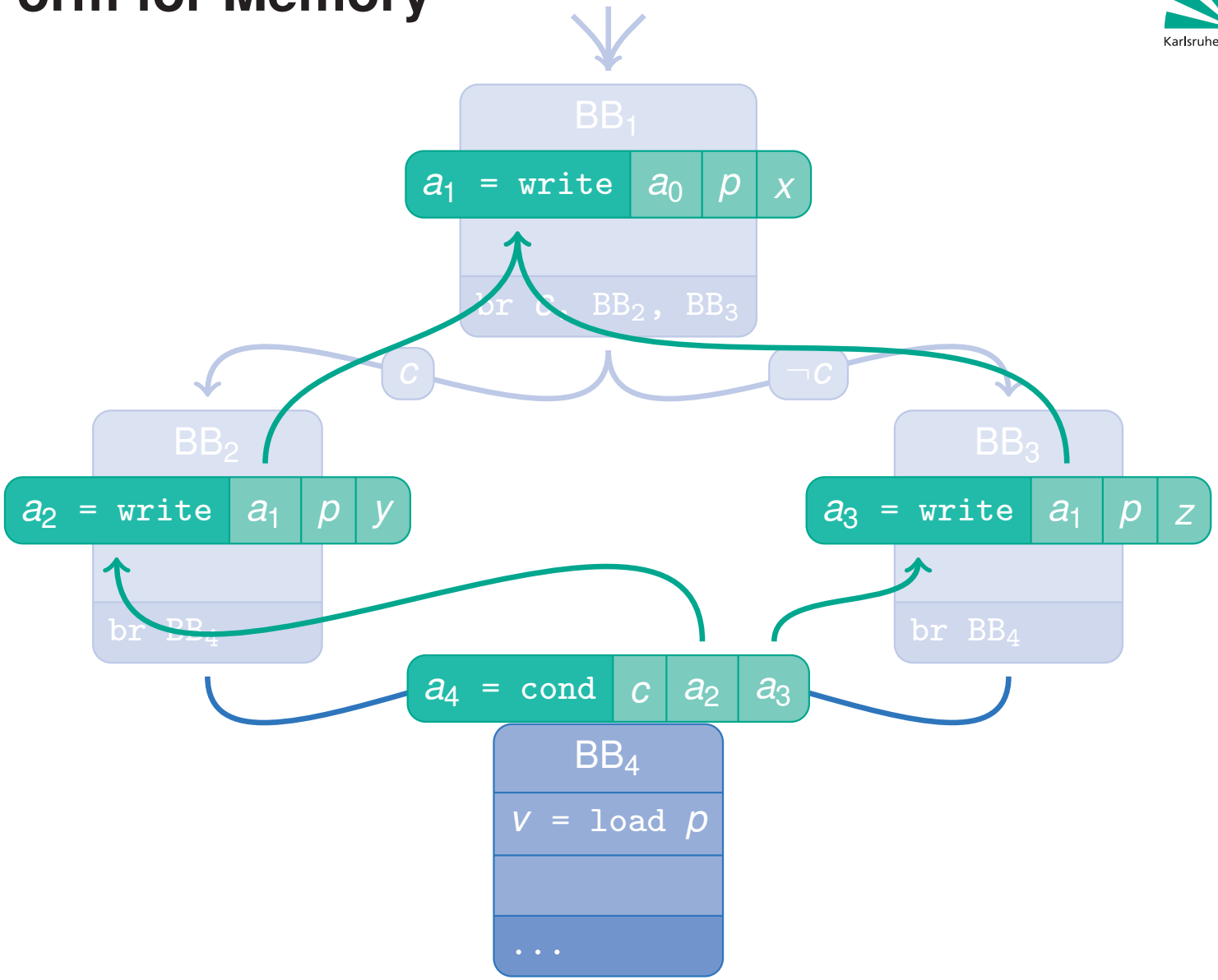
# SSA Form for Memory



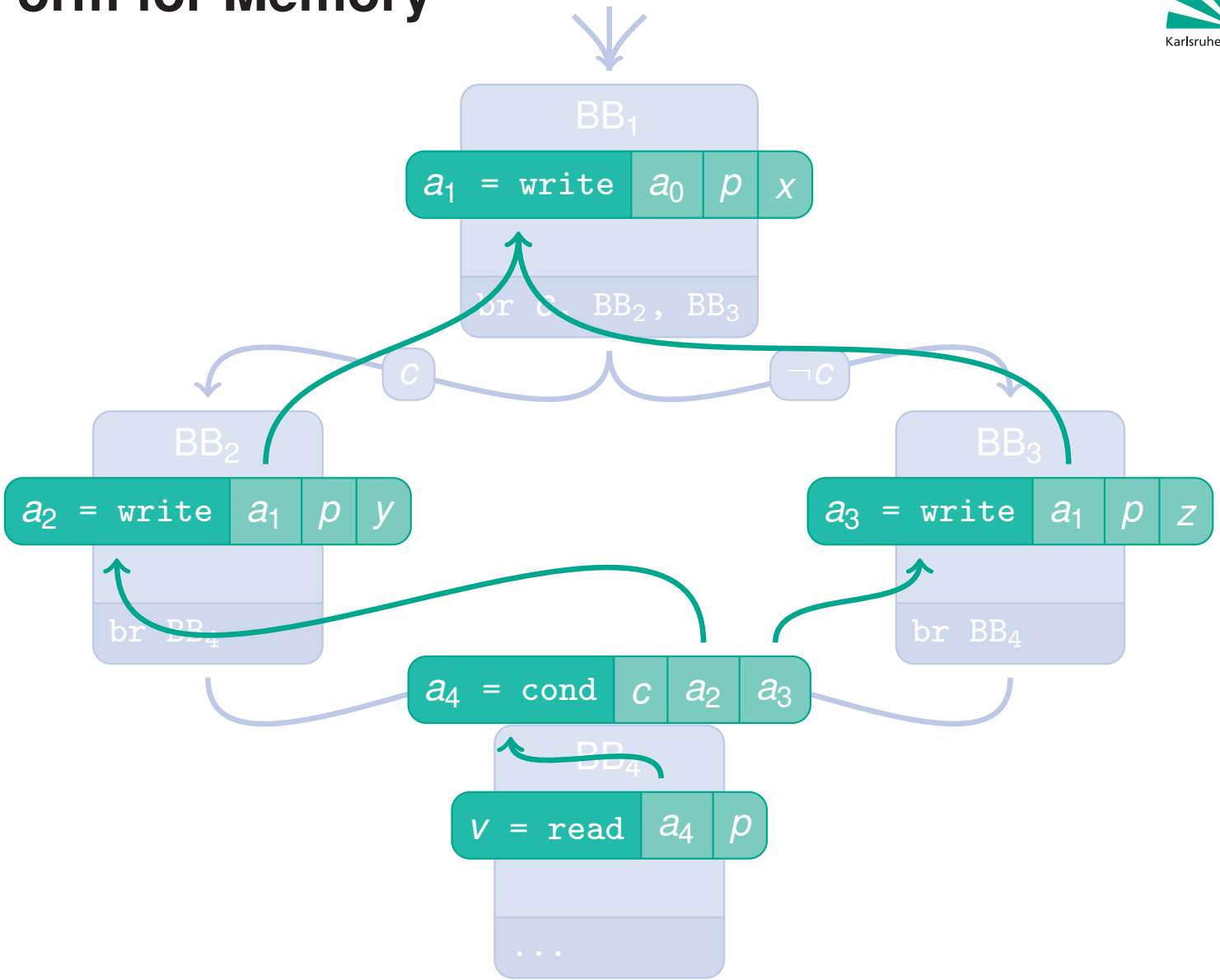
# SSA Form for Memory



# SSA Form for Memory



# SSA Form for Memory



# From Code to Logic

Goal: Show that

$$C \rightarrow P$$

To do:

Encode the desired properties $P$ ..... ☑

- Encoding the checks ..... ☑
- Guarding the checks ..... ☑

Encoding the source code $C$ ..... ☐

- SSA for calculations ..... ☑
- SSA across basic block boundaries ..... ☑
- Memory accesses ..... ☑
- Memory allocations/deallocations ..... ☐

# Memory Allocation Information

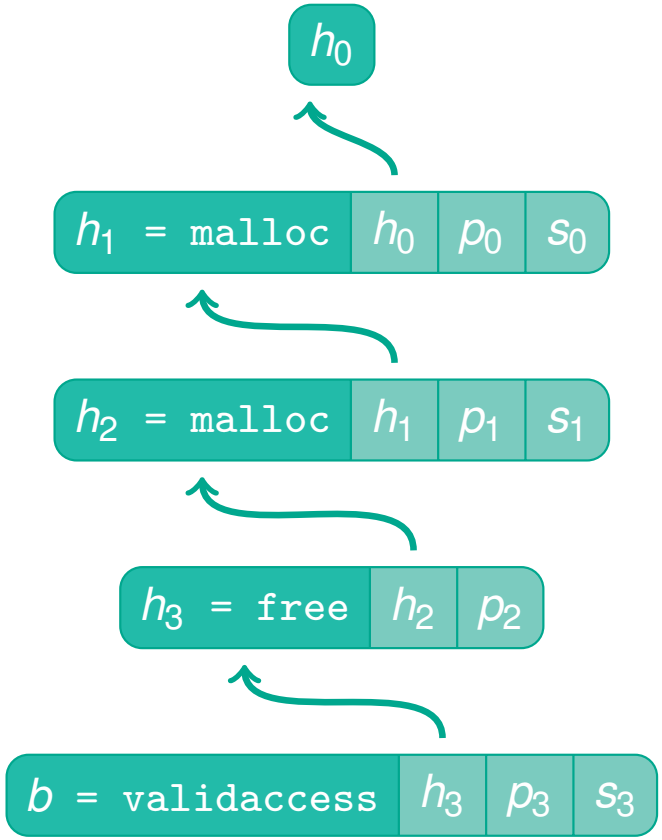
- Need access control for $T_a$
- Idea: Separate memory accesses and memory access validity
- $\Rightarrow$ Theory of heap allocations ($T_h$):

$$\mathit{malloc} : H \times I \times S \to H$$
$$\mathit{free} : H \times I \to H$$
$$\mathit{validaccess} : H \times I \times S \to \mathbb{B}$$

$H$ : Heap allocation state

$I$ : Heap index type

$S$ : Size type




$$\mathit{validaccess}(\mathit{free}(\mathit{malloc}(\mathit{malloc}(h_0, p_0, s_0), p_1, s_1), p_2), p_3, s_3)$$

# From Code to Logic

Goal: Show that

$$C \rightarrow P$$

To do:

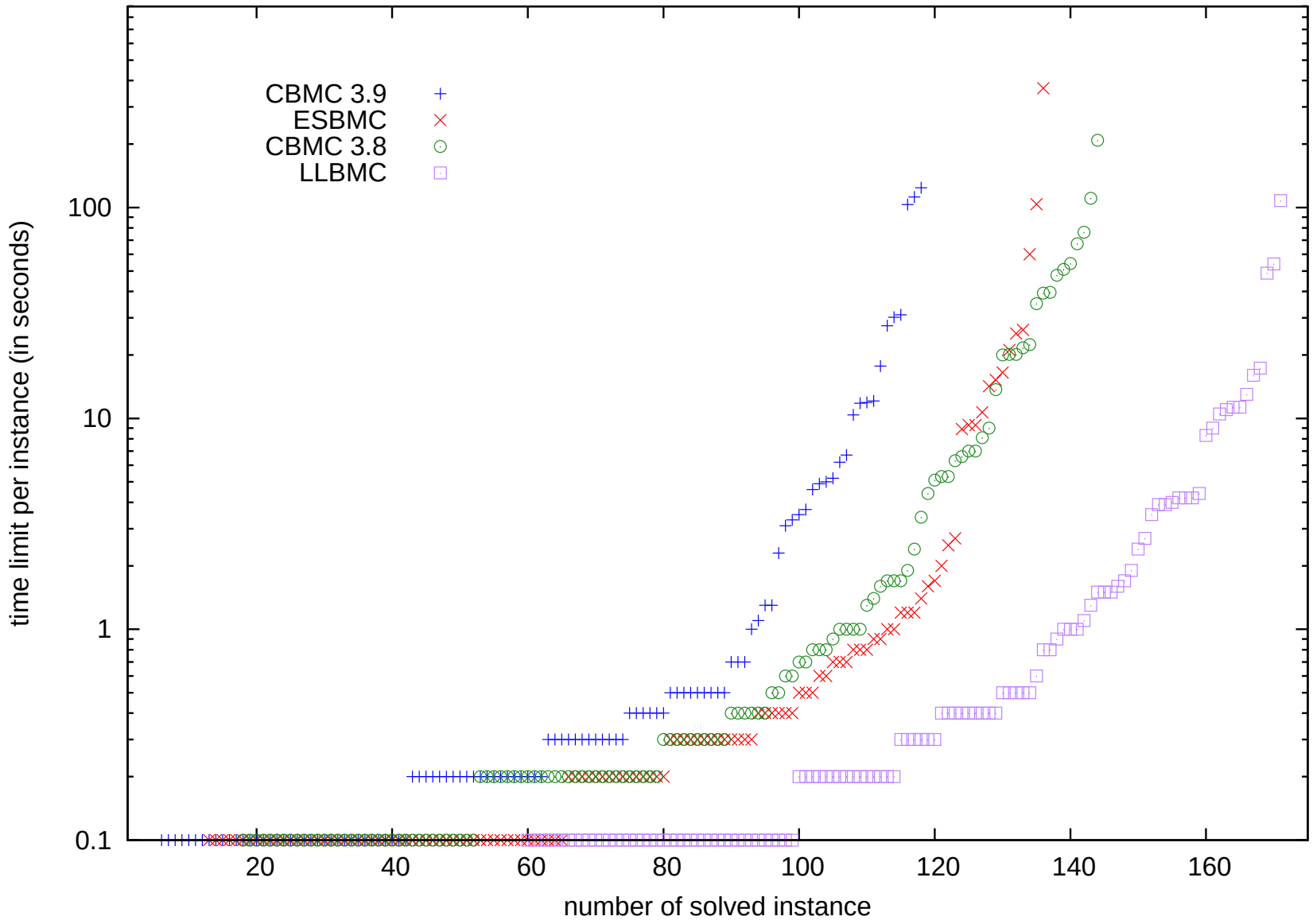
Encode the desired properties $P$ ☑

- Encoding the checks ☑
- Guarding the checks ☑

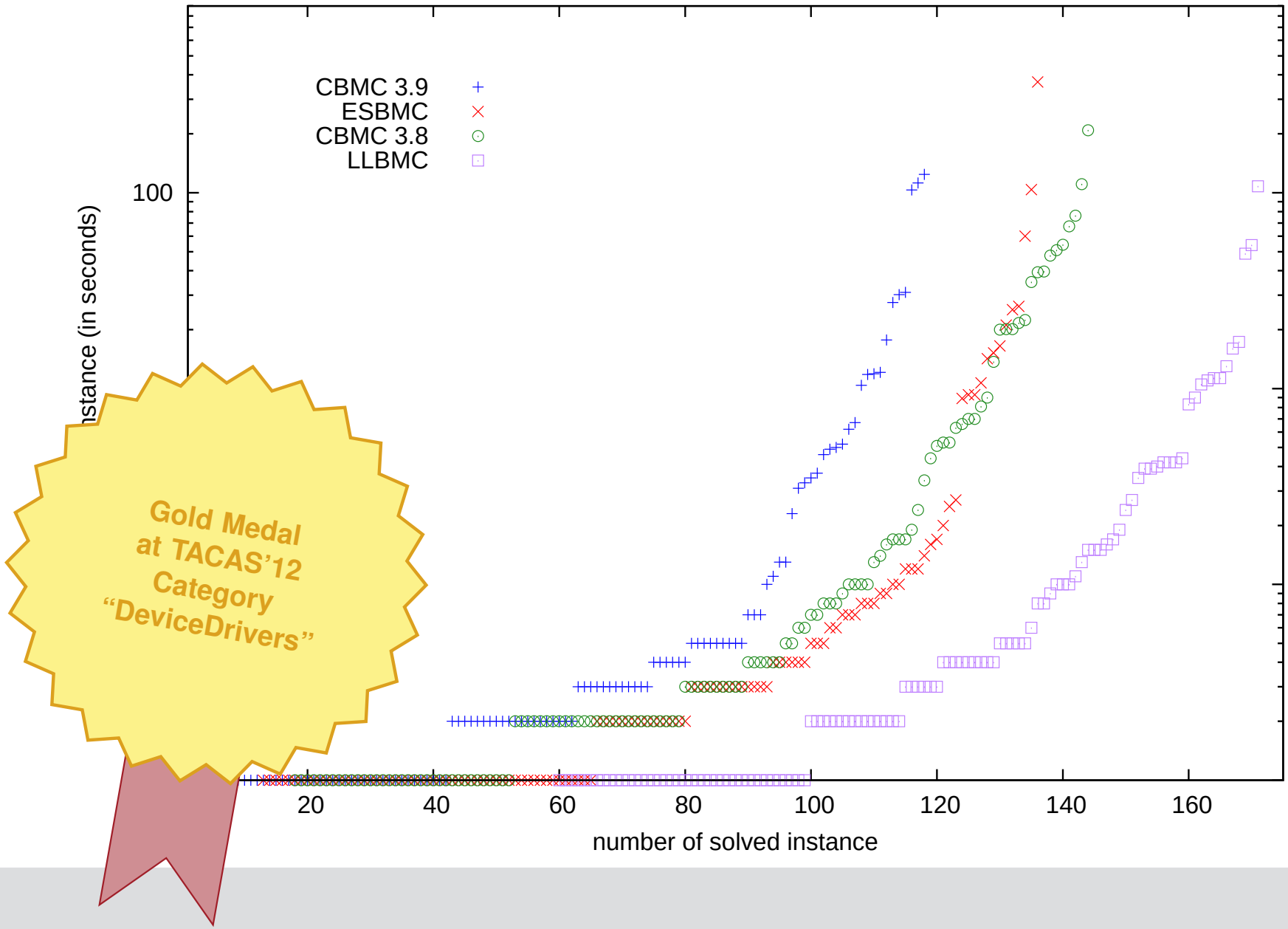
Encoding the source code $C$ ☑

- SSA for calculations ☑
- SSA across basic block boundaries ☑
- Memory accesses ☑
- Memory allocations/deallocations ☑

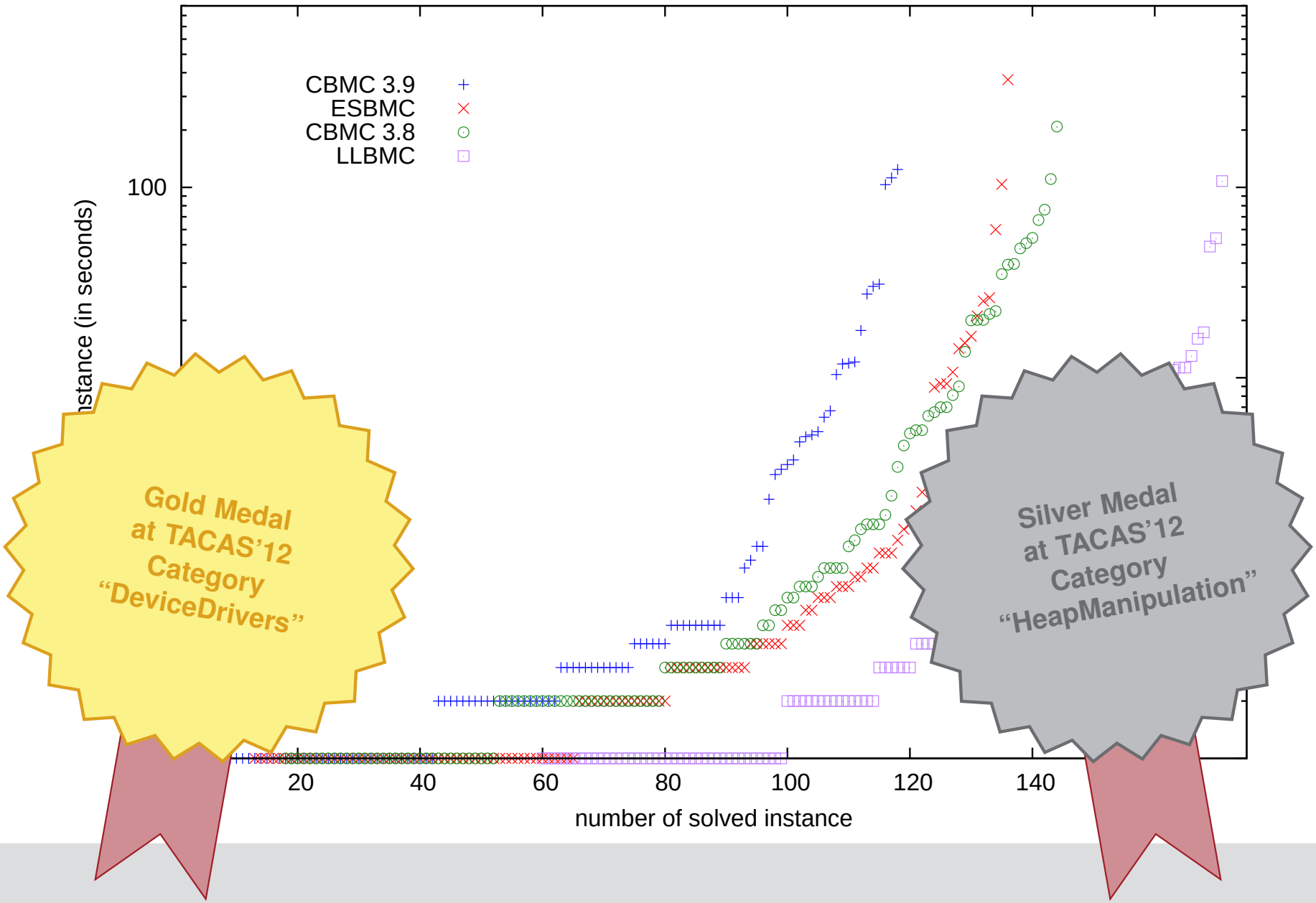
# Evaluation


CBMC 3.9
ESBMC
CBMC 3.8
LLBMC
time limit per instance (in seconds)
0.1
1
10
100
20
40
60
80
100
120
140
160
number of solved instance

# Evaluation


CBMC 3.9
ESBMC
CBMC 3.8
LLBMC
100
nstance (in seconds)
20
40
60
80
100
120
140
160
number of solved instance
Gold Medal
at TACAS'12
Category
"DeviceDrivers"

# Evaluation


CBMC 3.9
ESBMC
CBMC 3.8
LLBMC
100
nstance (in seconds)
20
40
60
80
100
120
140
number of solved instance
Gold Medal
at TACAS'12
Category
"DeviceDrivers"
Silver Medal
at TACAS'12
Category
"HeapManipulation"

# Outlook

- LLBMC for supporting the deductive verification process
- Modular bounded model checking
- Regression verification

# Thank you for listening

Karlsruhe Institute of Technology

http://llbmc.org