The CProver Suite of Verification Tools

Martin Brain
Peter Schrammel

UNIVERSITY OF OXFORD

Formal Methods 2016
Limassol, Cyprus

Rough Time Line

9:00 → 9:30 Introduction

9:30 → 10:00 CBMC work-along demonstration

10:00 → 10:30 Break

10:30 → 11:00 CBMC exercises

11:00 → 11:15 Concurrency and incremental BMC

11:15 → 11:30 Exercises

11:30 → 12:00 2LS

12:00 → 12:30 Exercises

It's Not Just Us...

University of Oxford Staff

- Martin Brain
- Lucas Cordeiro
- Cristina David
- Daniel Kroening
- Youcheng Sun

Researchers

- Peter Schrammel
- Michael Tautschnig
- Jade Alglave

PhD Students

- Dario Cattaruzza
- John Galea
- Sean Heelan
- Pascal Kesseli
- David Landsberg
- Lihao Liang
- Rajdeep Mukherjee
- Daniel Neville
- Daniel Poetzl
- Marcelo Sousa

diffblue

What Do You Want From This Tutorial?

What Do You Want From This Tutorial?

Goals:

1. What is available? What are solved problems? What are open problems?
2. Use tools for benchmarks, experiments, software development and teaching.
3. Interface and Integrate with CPROVER.

Overview

Part I

- Verification
- CPROVER by Example
- Interfaces
- Exercises

Part II

- Concurrency
- Incremental BMC
- Program Analysis beyond BMC

The Verification Question

The Key Question

Does (every run of) program P satisfy specification S?

The Verification Question

The Key Question

Does (every run of) program P satisfy specification S?

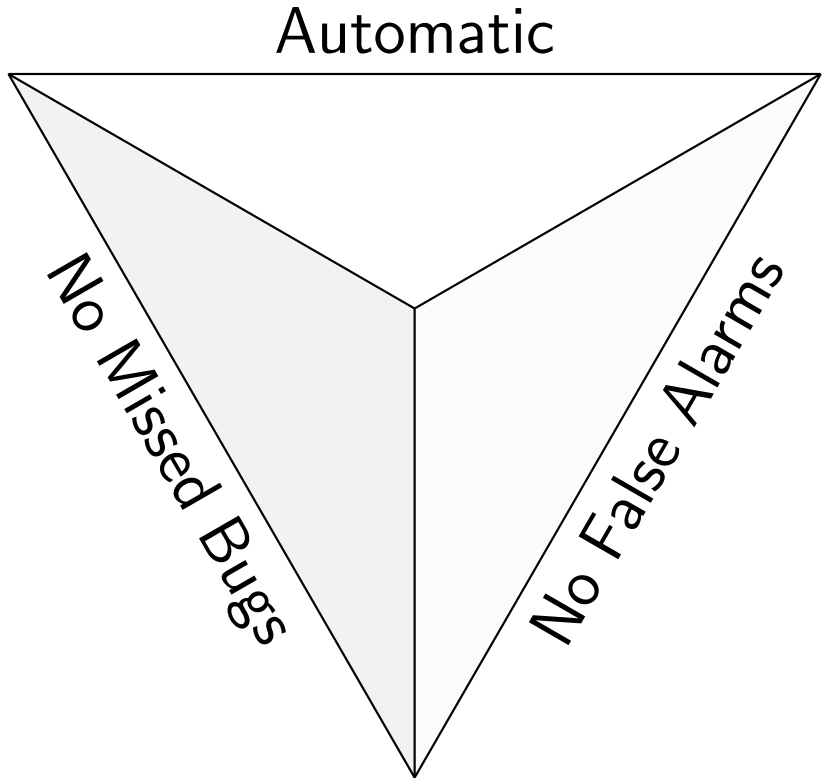
Program

- Sequential Software
- Concurrent Software
- Firmware
- Hardware
- Combinations

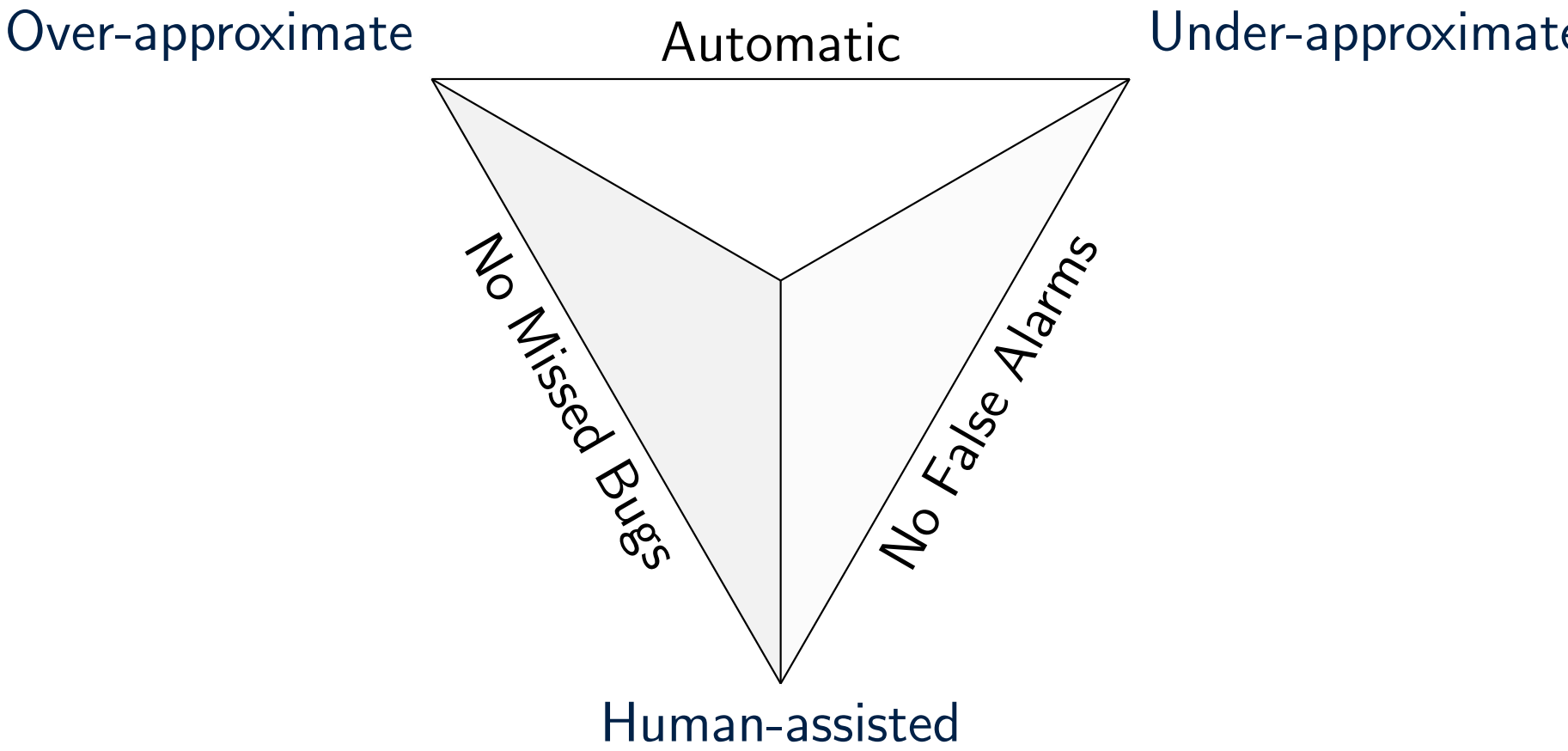
Specification

- Assertions
- Undefined behaviour
- Information flow
- Resources
- Termination / complexity

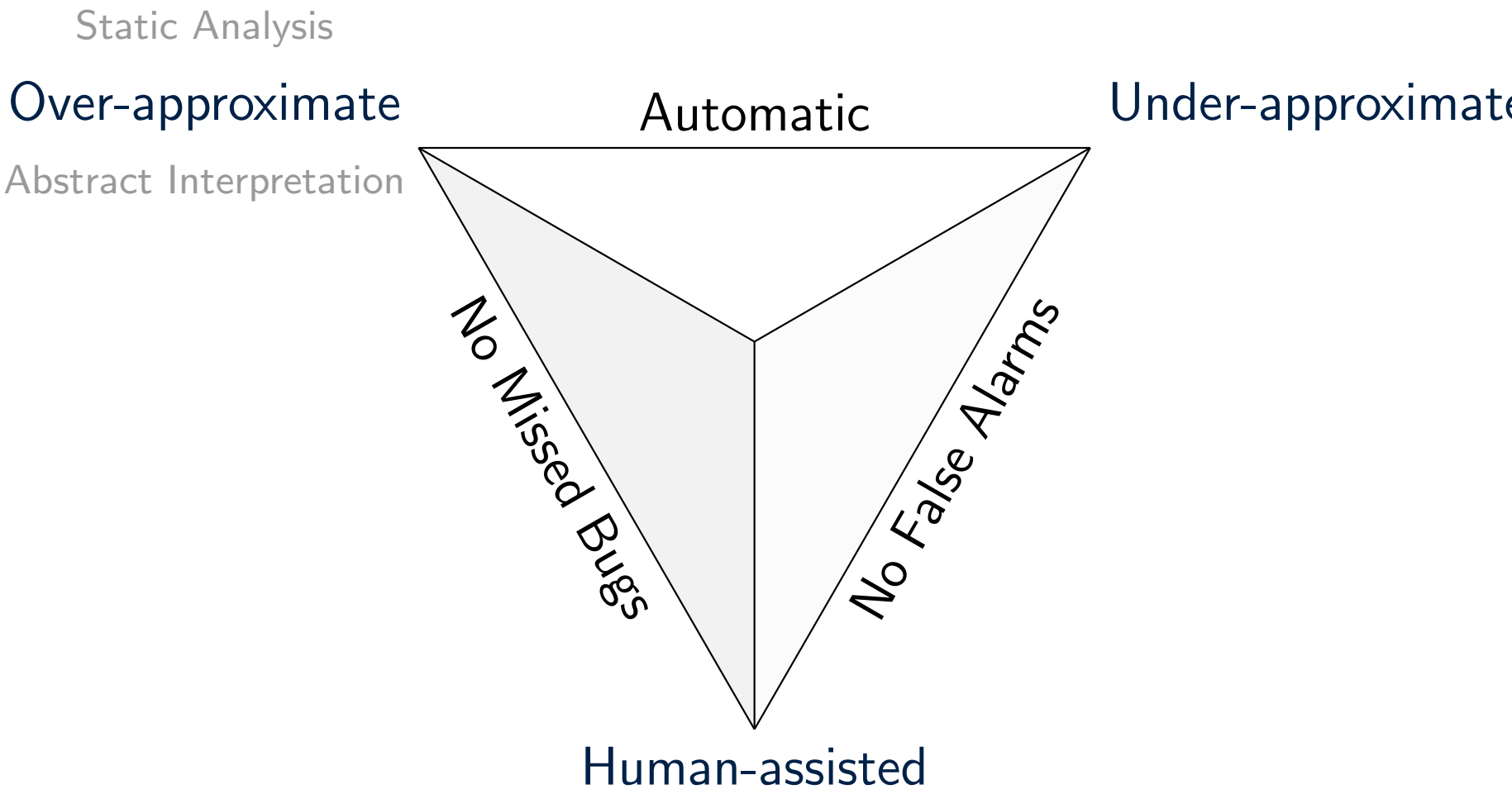
Martin's Pyramid Model of Verification



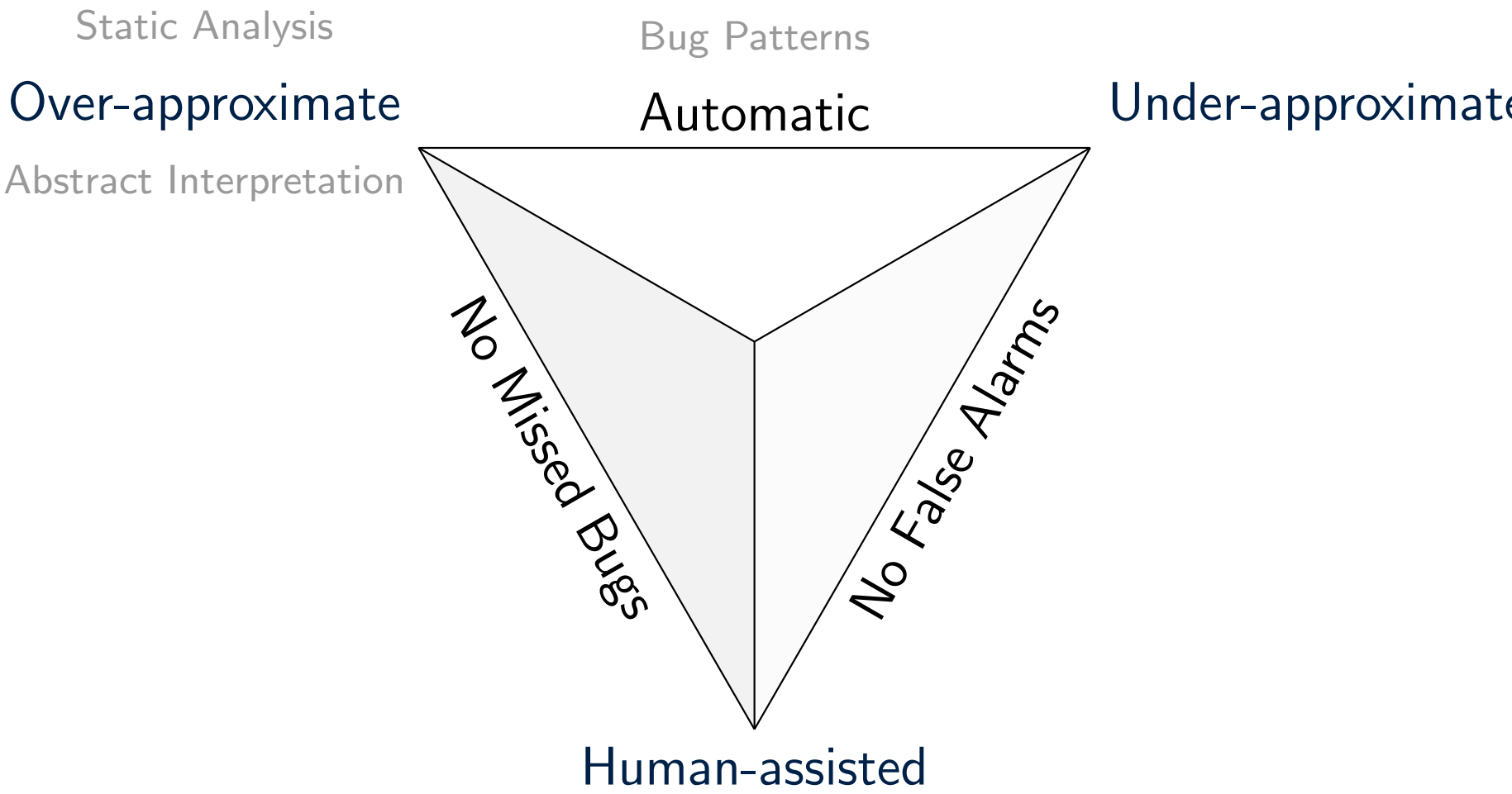
Martin's Pyramid Model of Verification



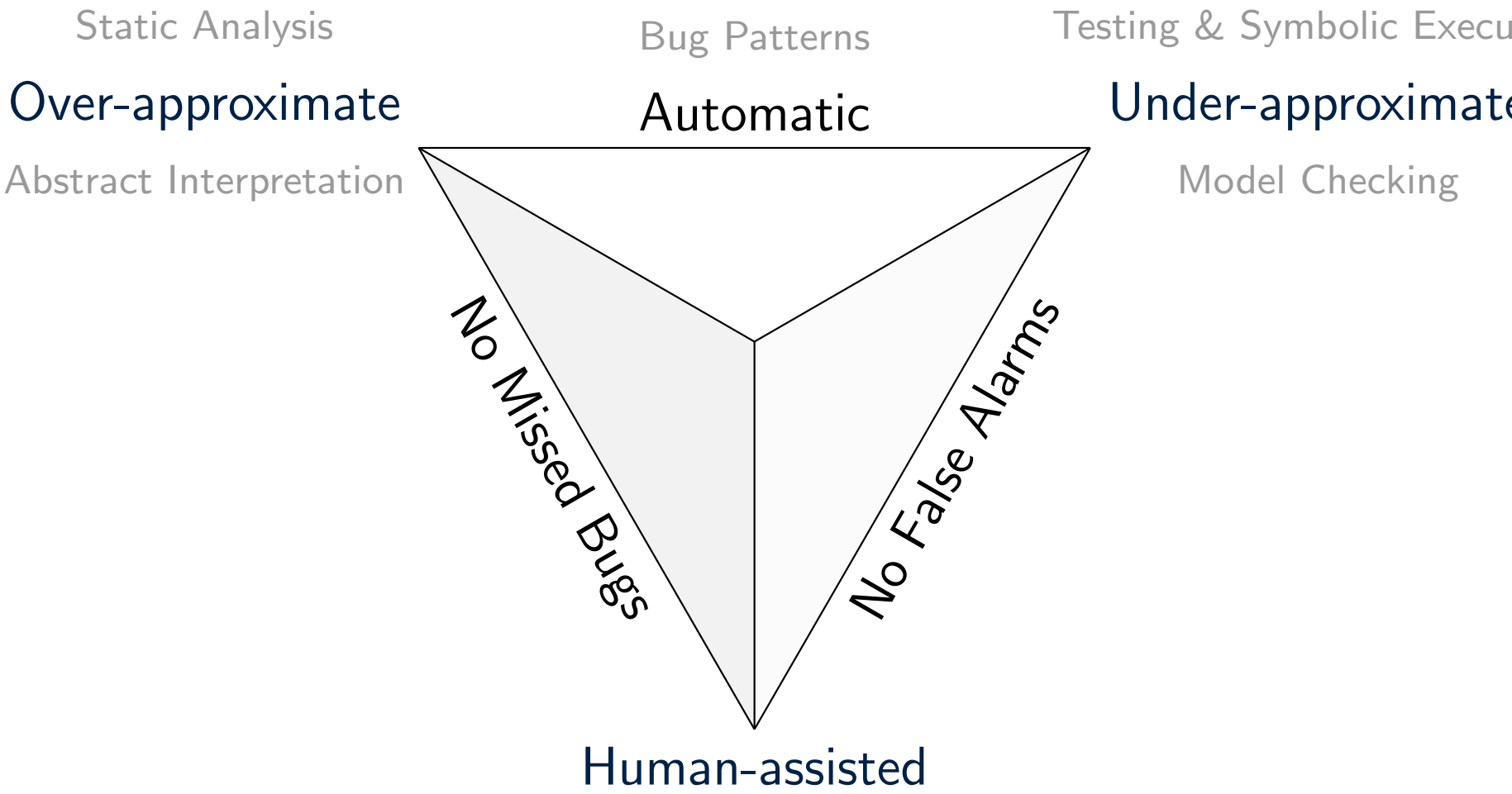
Martin's Pyramid Model of Verification



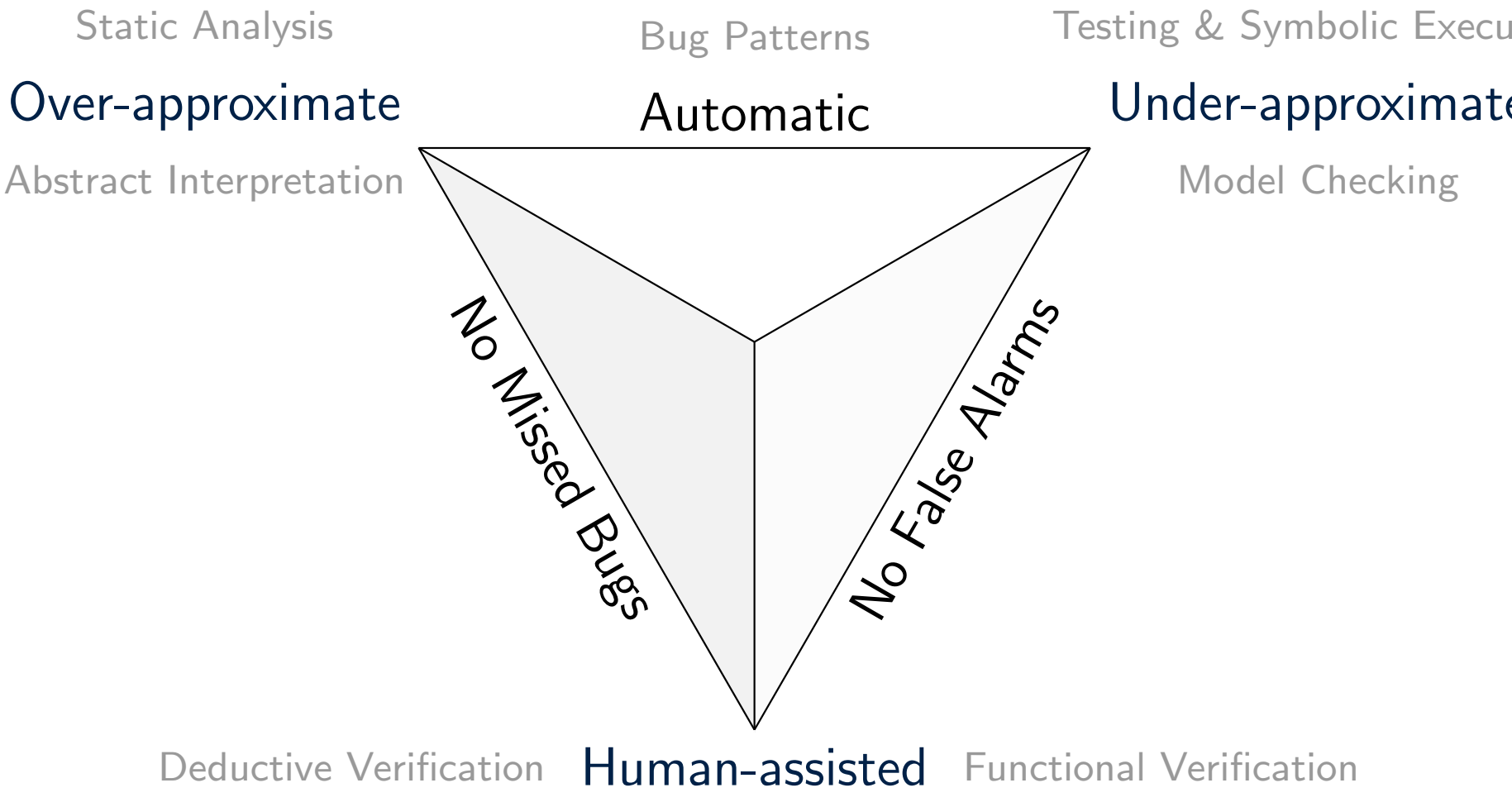
Martin's Pyramid Model of Verification



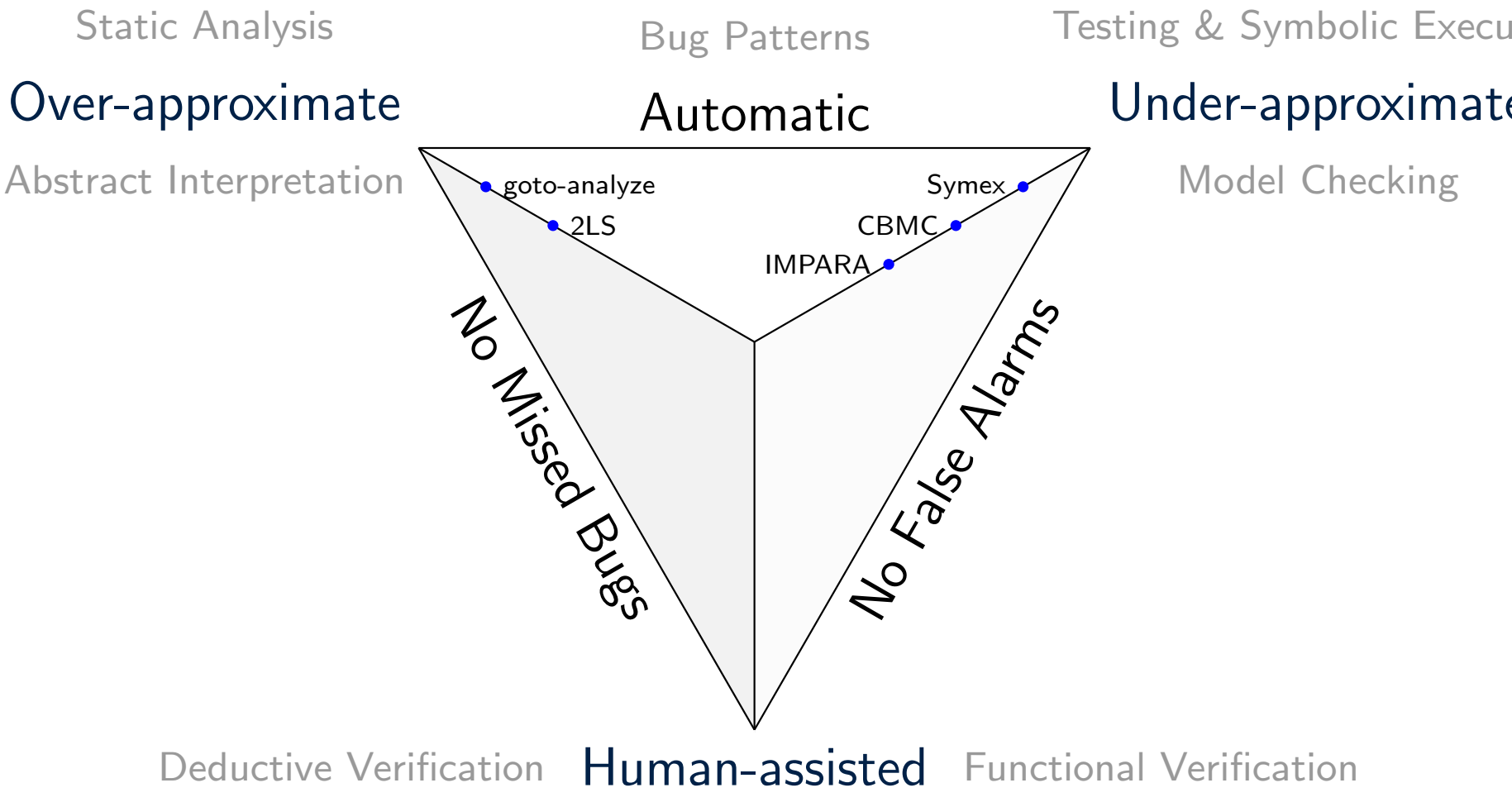
Martin's Pyramid Model of Verification



Martin's Pyramid Model of Verification



Martin's Pyramid Model of Verification



Use Formulae to Represent *Sets of States*

$$\text{Program} \rightarrow \text{Formula(e)}$$

Use Formulae to Represent *Sets of States*

$$\text{Program} \rightarrow \text{Formula(e)}$$

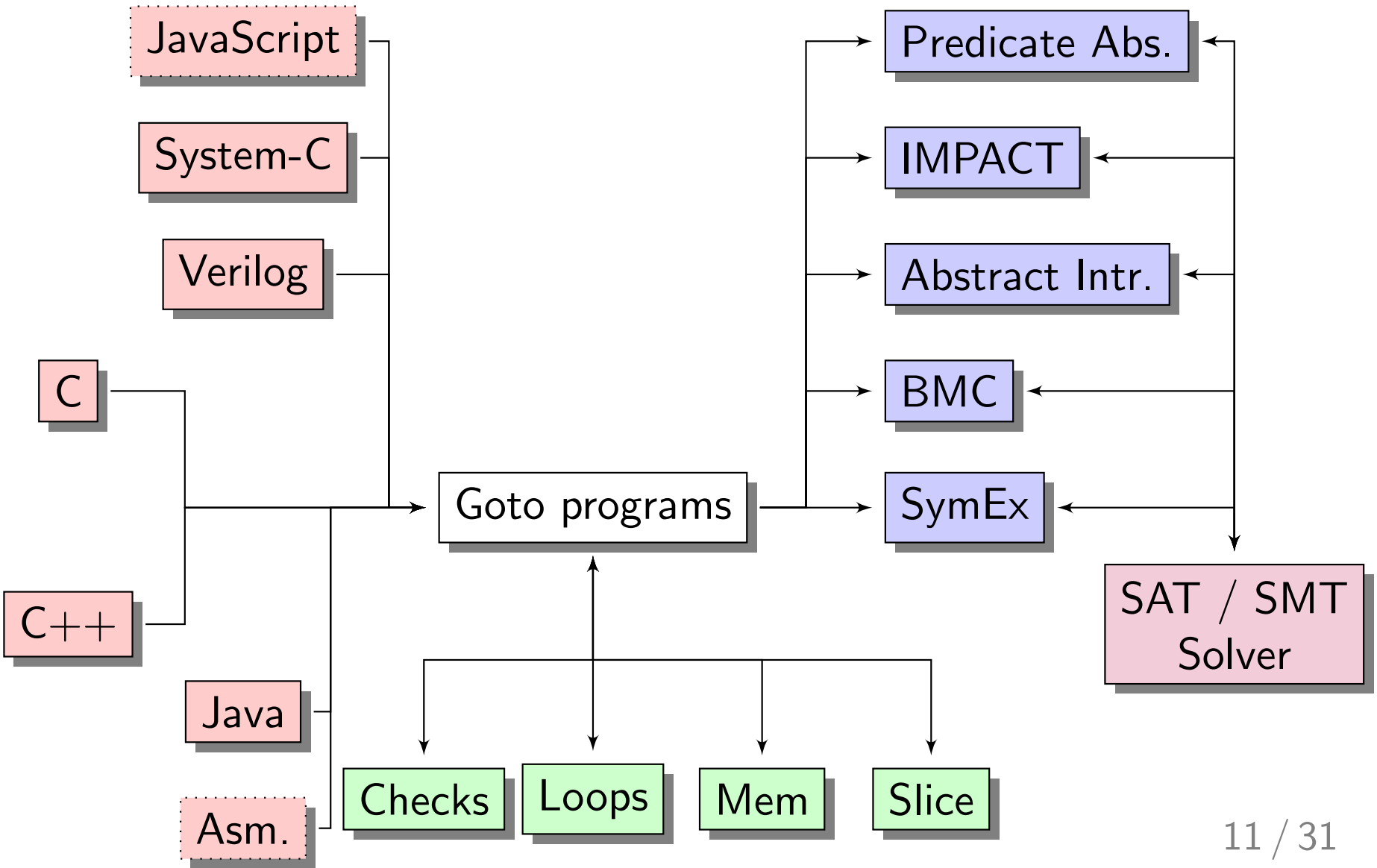
But what about the ...

- Strings
- Pointers
- Dynamic Memory
- Function Pointers
- Loops
- Recursion
- Concurrency
- Interrupts

The CPROVER Suite of Program Verification Tools

- A collection of tools built from a common code-base.
- Started in 2002/2003 with CBMC.
- Pioneered 'bit-blasting' for software verification.
- SV-COMP : 2014 Gold, 2015 Bronze
- Industrial users in automotive, EDA, aerospace.
- State-of-the-art support for floating-point, multi-core, firmware, ...

CPROVER System Architecture



Install CBMC Now!

Binary:

`http://www.cprover.org/cbmc/`

Source:

`http://github.com/diffblue/cbmc/`

Examples:

`http://www.cprover.org/tutorial/`

abs.c

```
#include <assert.h>
#include <stdlib.h>

int abs (int x) {
  int y = x;

  if (x < 0) {
    y = -x;
  }

  return y;
}
```

1. Front End

```
cbmc abs.c --function abs --show-goto-functions
```

1. Front End

```
cbmc abs.c --function abs --show-goto-functions
```

Notice:

- Uses system preprocessor and headers
- Library functions are stubs
- OS/Architecture specific information

2. Instrumentation

```
cbmc abs.c --function abs --signed-overflow-check
           --show-goto-functions
```

2. Instrumentation

```
cbmc abs.c --function abs --signed-overflow-check
           --show-goto-functions
```

Notice:

- `--*-check` flags.
- Specifications are assertions.

3.A. Static Single Assign

```
#include <assert.h>
#include <stdlib.h>

int abs (int x) {
  int y = x;

  if (x < 0) {
    y = -x;
  }

  return y;
}
```

```
#include <assert.h>
#include <stdlib.h>

int abs (int x1) {
  int y2 = x1;

  int guard1 = (x1 < 0);
    int y3 = -x1;
  int y4 = (guard1) ? y3 : y2

  return y4;
}
```

3.A. Static Single Assign

```
#include <assert.h>
#include <stdlib.h>

int abs (int x) {
  int y = x;

  if (x < 0) {
    y = -x;
  }

  return y;
}
```

```
#include <assert.h>
#include <stdlib.h>

int abs (int x1) {
  int y2 = x1;

  int guard1 = (x1 < 0);
    int y3 = -x1;
  int y4 = (guard1) ? y3 : y2

  return y4;
}
```

```
cbmc abs.c --function abs --signed-overflow-check
           --program-only
```

3.B. Verification Conditions

```
cbmc abs.c --function abs --signed-overflow-check
           --show-vcc
```

3.B. Verification Conditions

```
cbmc abs.c --function abs --signed-overflow-check
           --show-vcc
```

Notice:

- `--slice-formula`
- Each assertion becomes a VC.

Putting It All Together

```
cbmc abs.c --function abs --signed-overflow-check
```

Putting It All Together

```
cbmc abs.c --function abs --signed-overflow-check
```

Notice:

- Instructions in goto-program $\rightarrow$ steps in trace.
- Reporting values and locations.

escapeFunction.c

```
#include <string.h>
#include <stdlib.h>

char * escape (const char *s) {
  char * escaped = malloc((strlen(s) + 1) * 2);
  char * output = escaped;

  *(output++) = '"';
  while (*s != '\0') {
    if ((*s == '"') || (*s == '\\')) {
      *(output++) = '\\';
    }
    *(output++) = *(s++);
  }
  *(output++) = '"';
  *(output++) = '\0';

  return escaped;
}
```

Harnesses and Unwinding

```
cbmc escapeFunction.c --function escape
--pointer-check --bounds-check --slice-formula
```

Harnesses and Unwinding

```
cbmc escapeFunction.c --function escape
--pointer-check --bounds-check --slice-formula
```

```
cbmc escapeFunction.c --function escape
--pointer-check --bounds-check --slice-formula
                    --unwind 4
```

escapeFunctionHarness.c

```
#include "escapeFunction.c"

char nondet_char (void);

int main (void) {
  char string[16];
  int i;

  for (i = 0; i < 16; ++i) {
    string[i] = nondet_char();
    if (string[i] == '\0') { break; }
  }

  char *output = escape(string);

  return 0;
}
```

Unwinding Assertions

```
cbmc escapeFunctionHarness.c --pointer-check
  --bounds-check --slice-formula --unwind 4
        --no-unwinding-assertions
```

goto-cc

```
goto-cc simple.c -o simple.goto
```

goto-cc

```
goto-cc simple.c -o simple.goto
```

Notice:

- Compiler replacement for build system integration.
- Has both compile and link phases.
- Also `goto-cl`, `goto-cw`, ...

goto-instrument

```
goto-instrument --dump-c simple.goto
```

goto-instrument

```
goto-instrument --dump-c simple.goto
```

Notice:

- Program to program transforms
- k-induction
- loop-acceleration

Program → Result

XML

`--xml-ui`

`--xml-interface`

SV-COMP GraphML

`--graphml-cex`

(Counter-examples only)

Program → Formula

DIMACS

```
cbmc abs.c --function abs --signed-overflow-check
                          --dimacs
```

SMT-LIB

```
cbmc abs.c --function abs --signed-overflow-check
                    --smt2 --outfile -

cbmc abs.c --function abs --signed-overflow-check
                  --smt2 --outfile - --fpa
```

Internal Interfaces

Goto-Programs

`goto-programs/goto_program_template.h : instructiont`

Formulae

`util/decision_procedure.h : decision_proceduret`

`solvers/prop/prop_conv.h : prop_convt`

Challenges and The Future

We Always Need Better...

- Provers
- Pointer analysis
- Analysis Techniques

How Do We ...

- Loop Acceleration
- Summaries
- Handle libraries
- Strings

Someone Could Build...

Front-ends x86, PHP, Python, ...

Instrumentation `dump-*`

Back-ends Interpolation

Why Not Try...

- Crypto libraries
- Parts of the Linux Kernel
- Trust Zone OS
- GLibc

Try These!

1. Fix the error in `abs.c` and confirm that it is fixed.
2. Using `escapeFunctionHarness.c`:
   1. Fix the off-by-one-error in allocation and confirm it is fixed.
   2. By increasing the unwinding limit, locate and fix the other bug.
3. `loopUnwinding.c`
4. `sorting.c`
5. `searchTree.c`

`http://www.cprover.org/tutorial/`