Sequoll: a framework for model checking binaries

NICTA

Bernard Blackham, Gernot Heiser

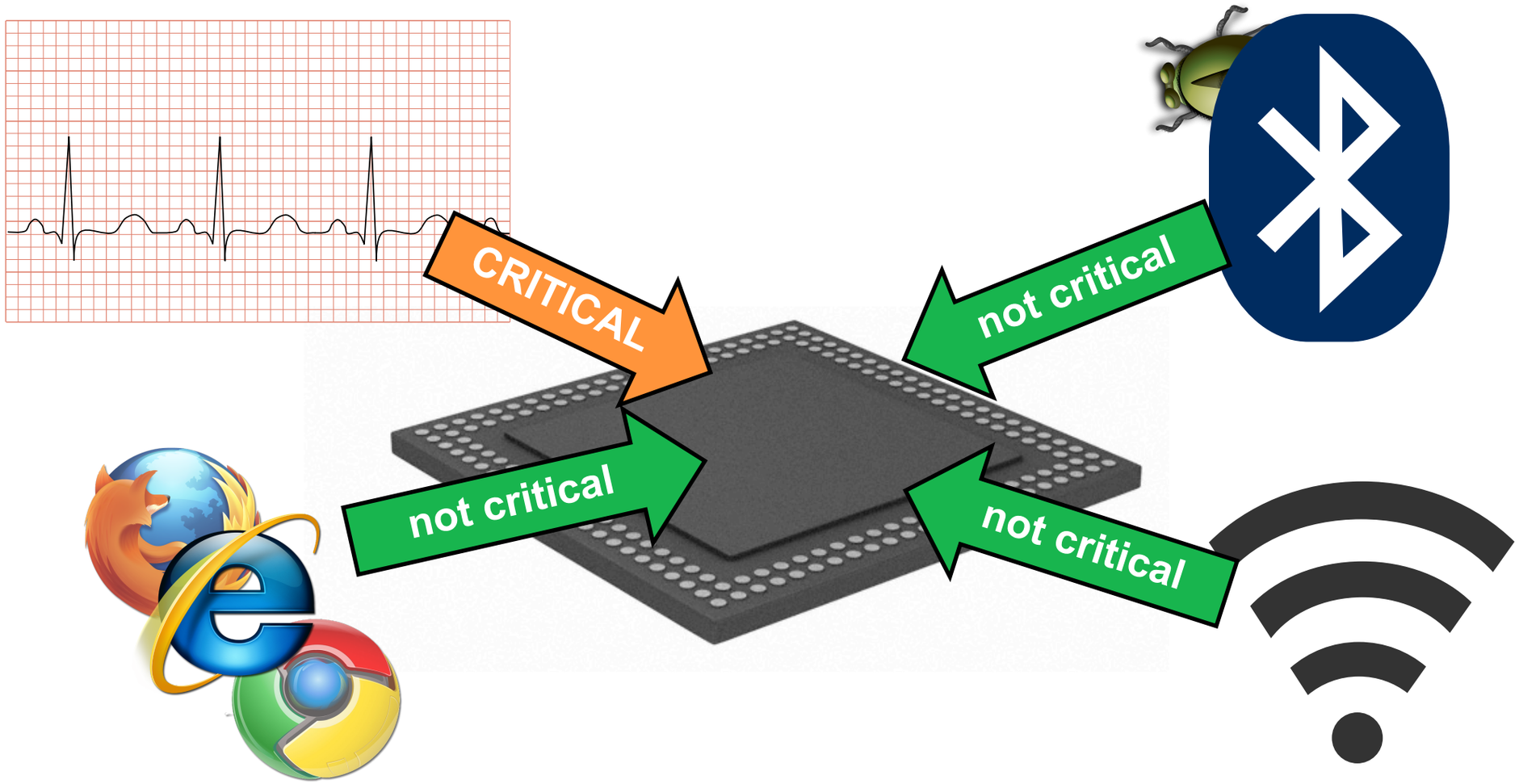
School of Computer Science & Engineering, UNSW
Software Systems Research Group, NICTA
Sydney, Australia

Australian Government
Department of Broadband, Communications and the Digital Economy
Australian Research Council

NICTA Funding and Supporting Members and Partners

Motivation

- The desire for a *trustworthy* kernel to build reliable **mixed-criticality real-time** systems



Motivation

- The desire for a *trustworthy* kernel to build reliable **multi-criticality real-time** systems

- Using **seL4** to guarantee:
  - **functional correctness** through formal proof

    (Klein et al., SOSP 2009)
  - **timing constraints** through sound WCET analysis

    (Blackham et al., RTSS 2011)

Motivation

NICTA

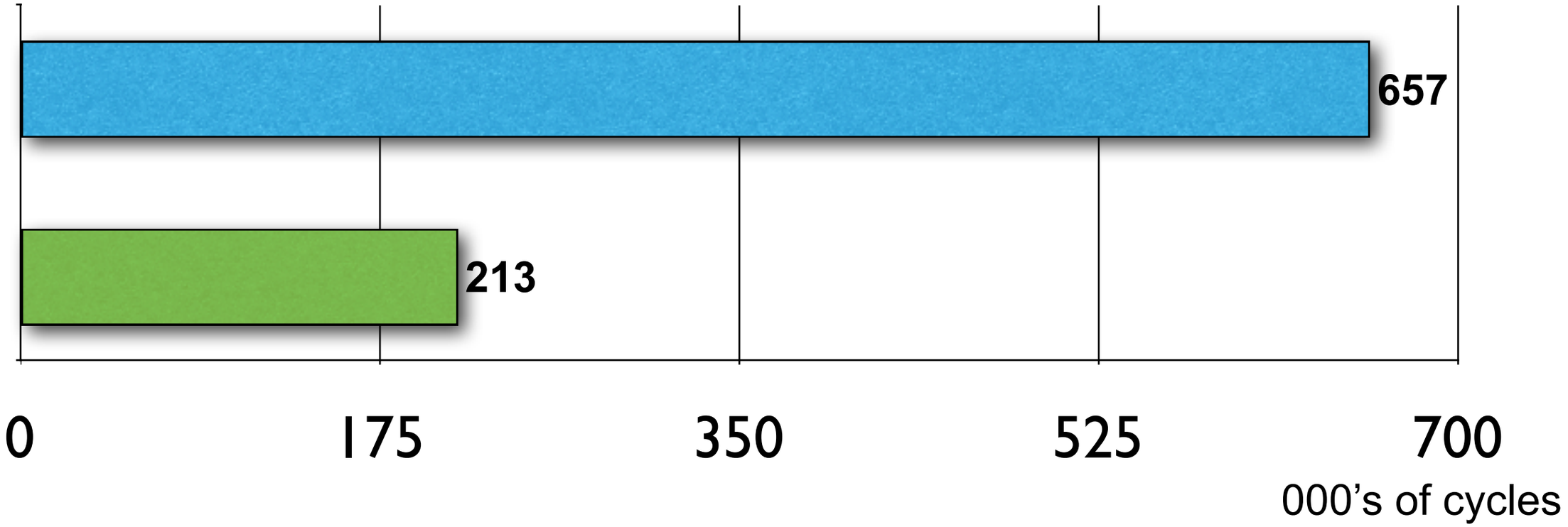
- Current analysis uses annotations to specify:
  - loop counts
  - infeasible paths

- We want to reduce scope for errors in WCET analysis.

Results

NICTA

Estimated worst-case execution time of seL4



seL4 is large

- Small by microkernel standards
- Large by WCET standards

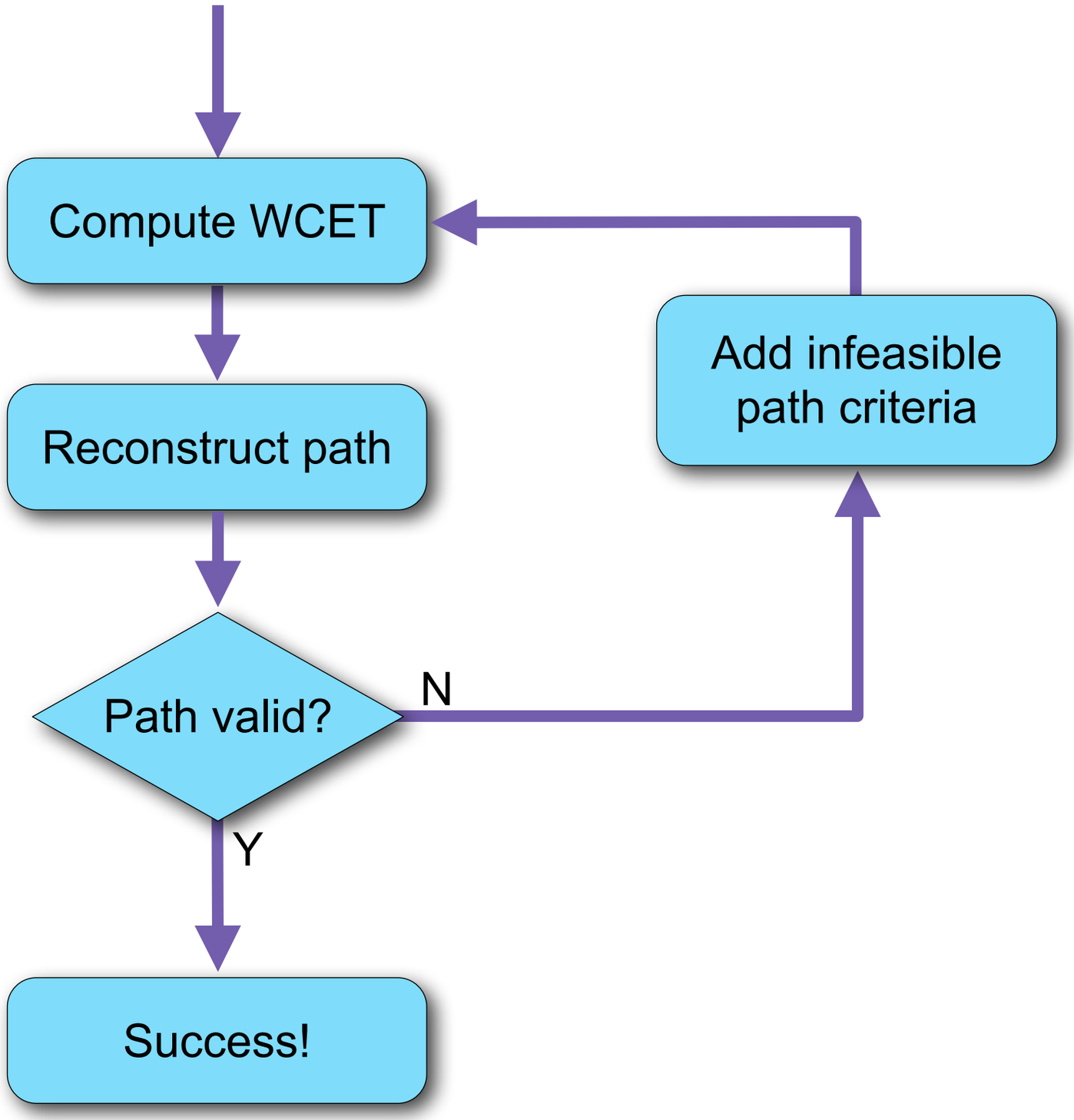
C source

~8,700 lines
316 functions
76 loops

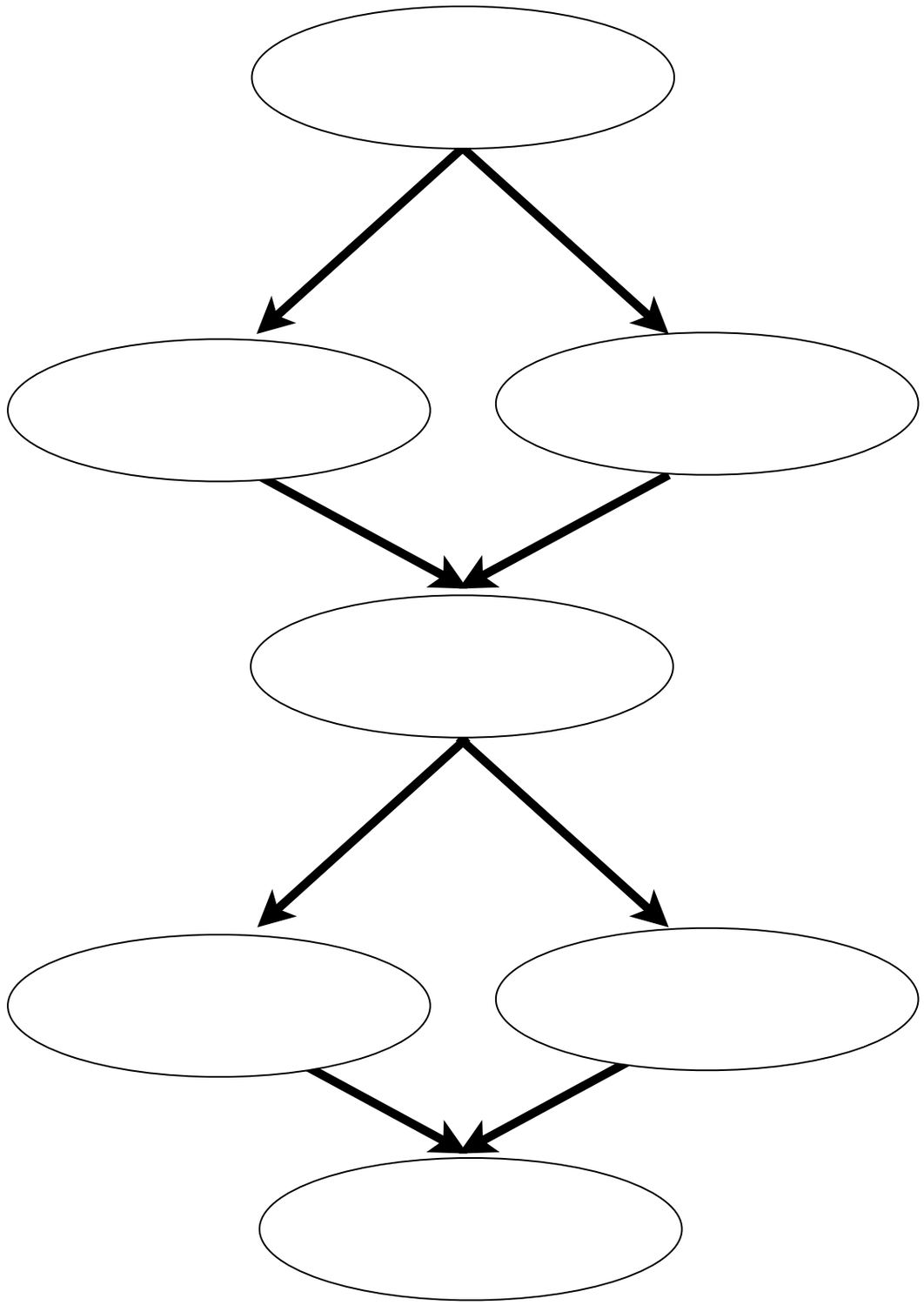
Binary (ARM)

~10,000 instructions
228 functions
56 loops
2,384 basic blocks
~400,000 basic blocks when inlined

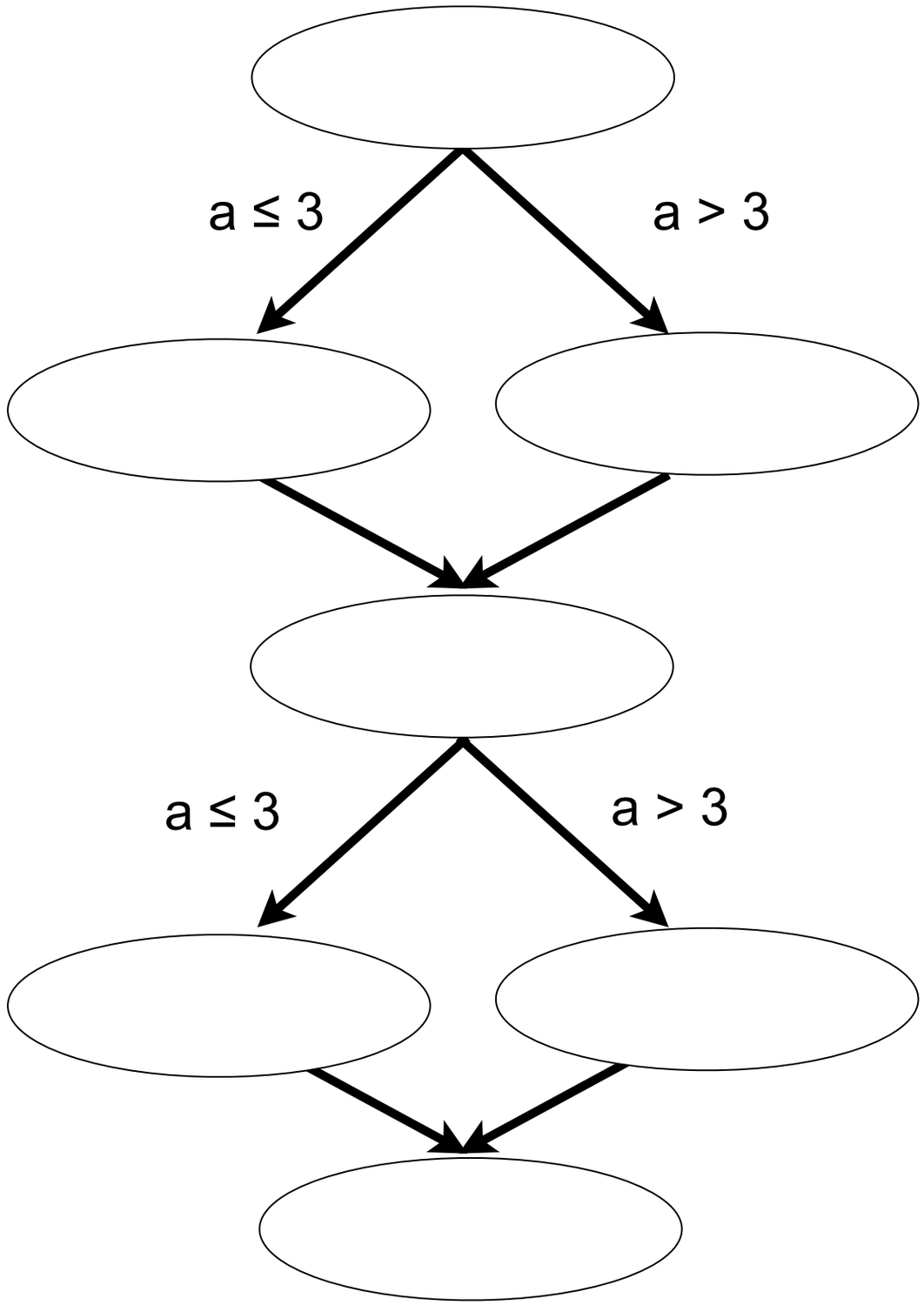
WCET computation process



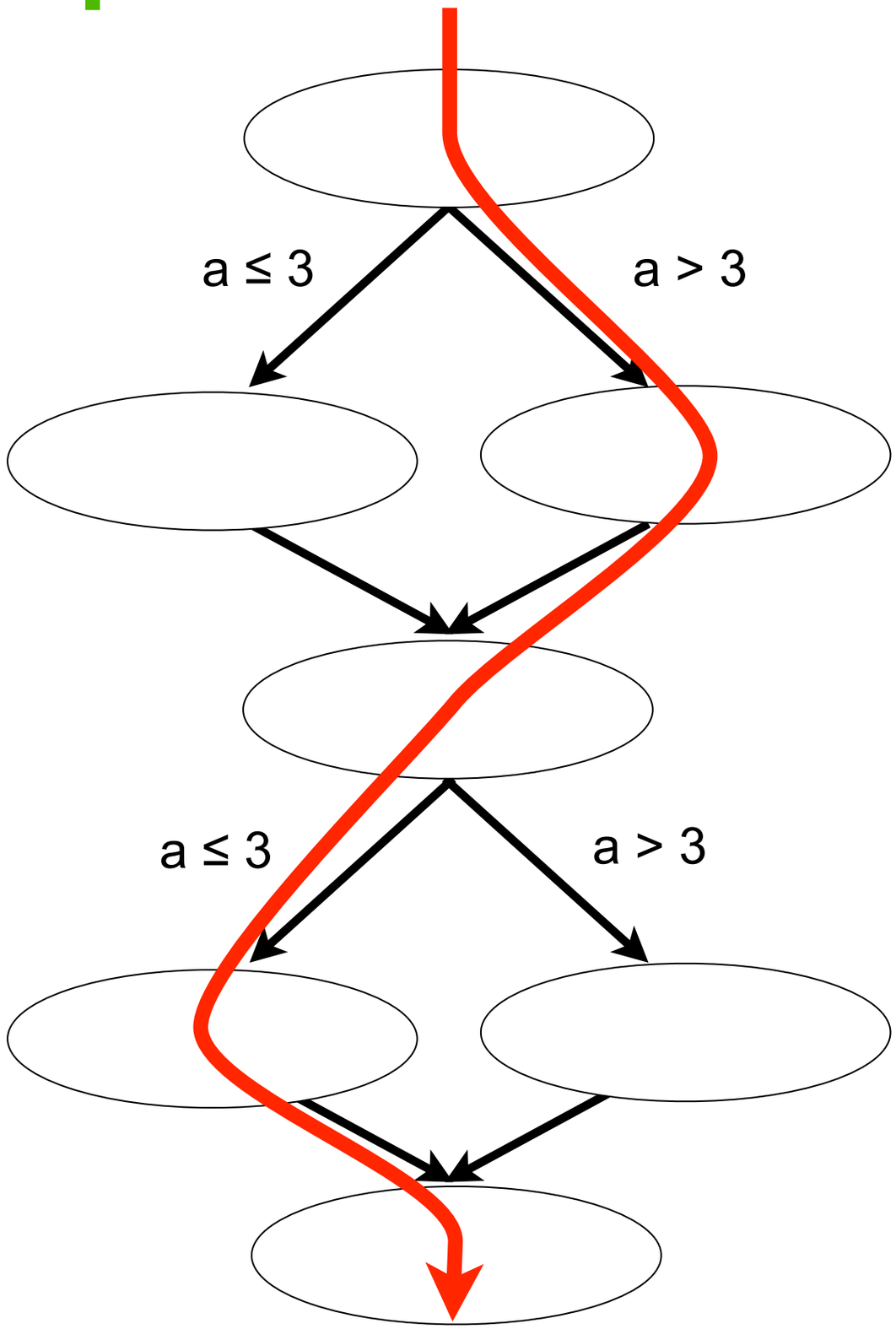
Infeasible paths



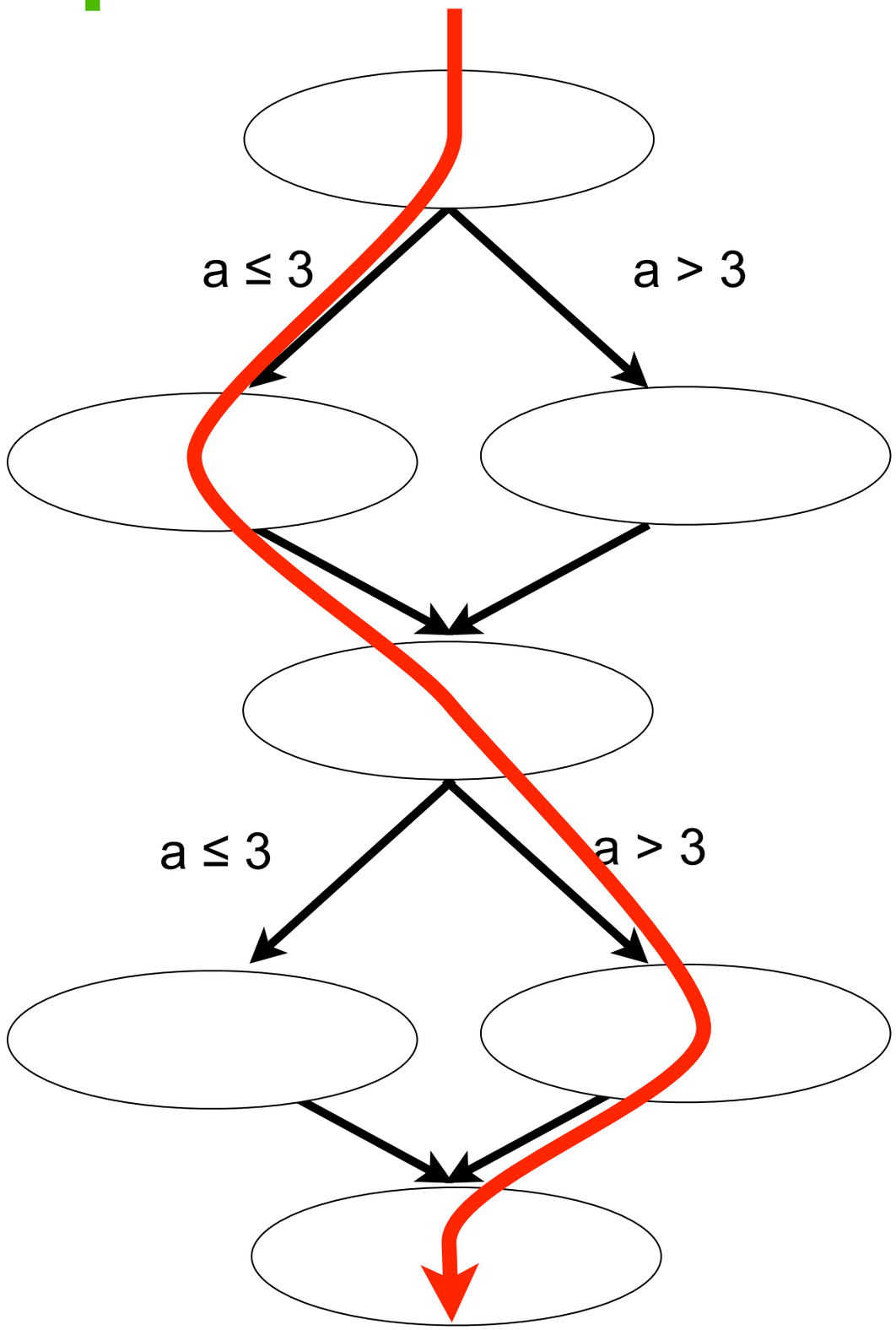
Infeasible paths



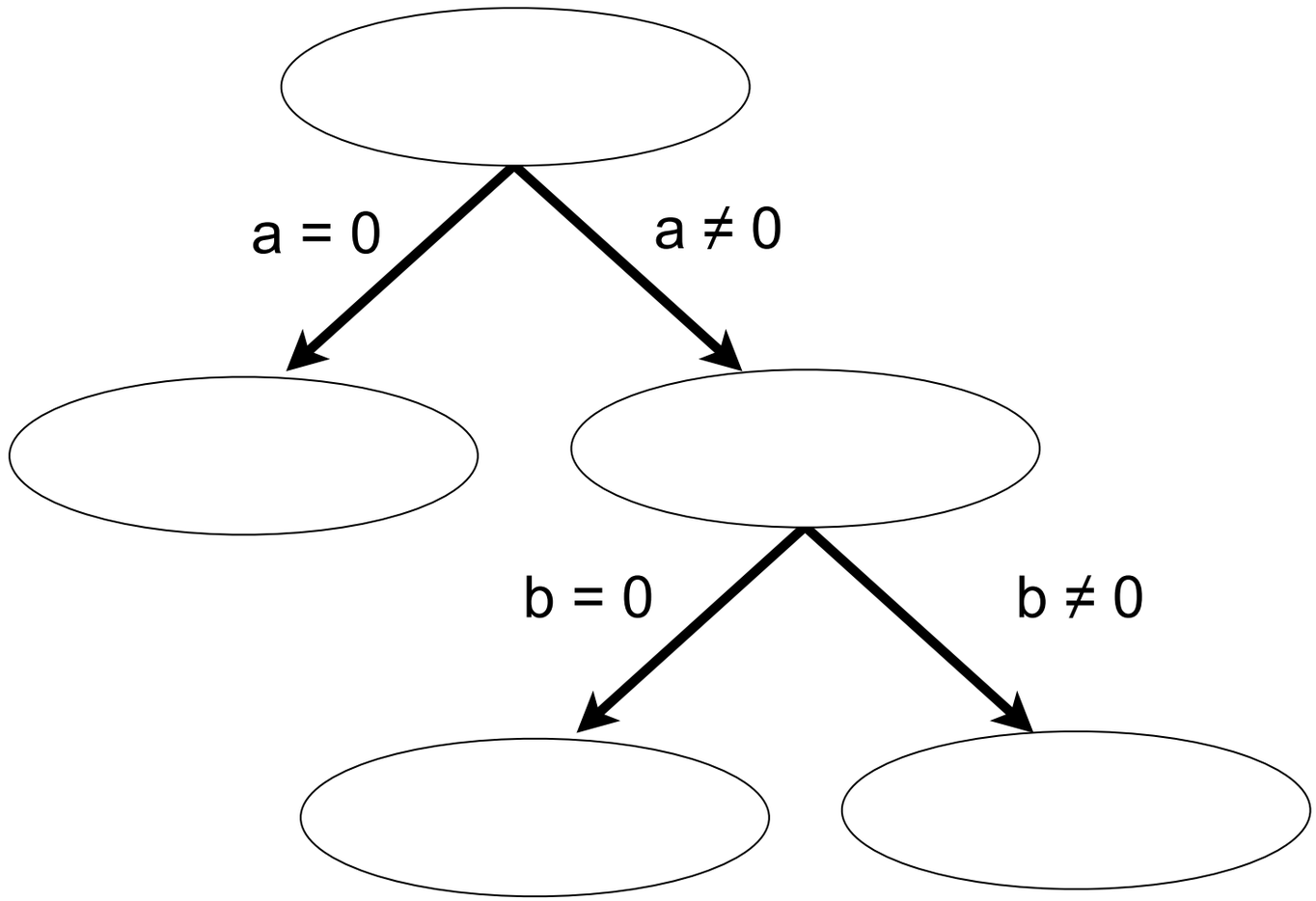
Infeasible paths



Infeasible paths

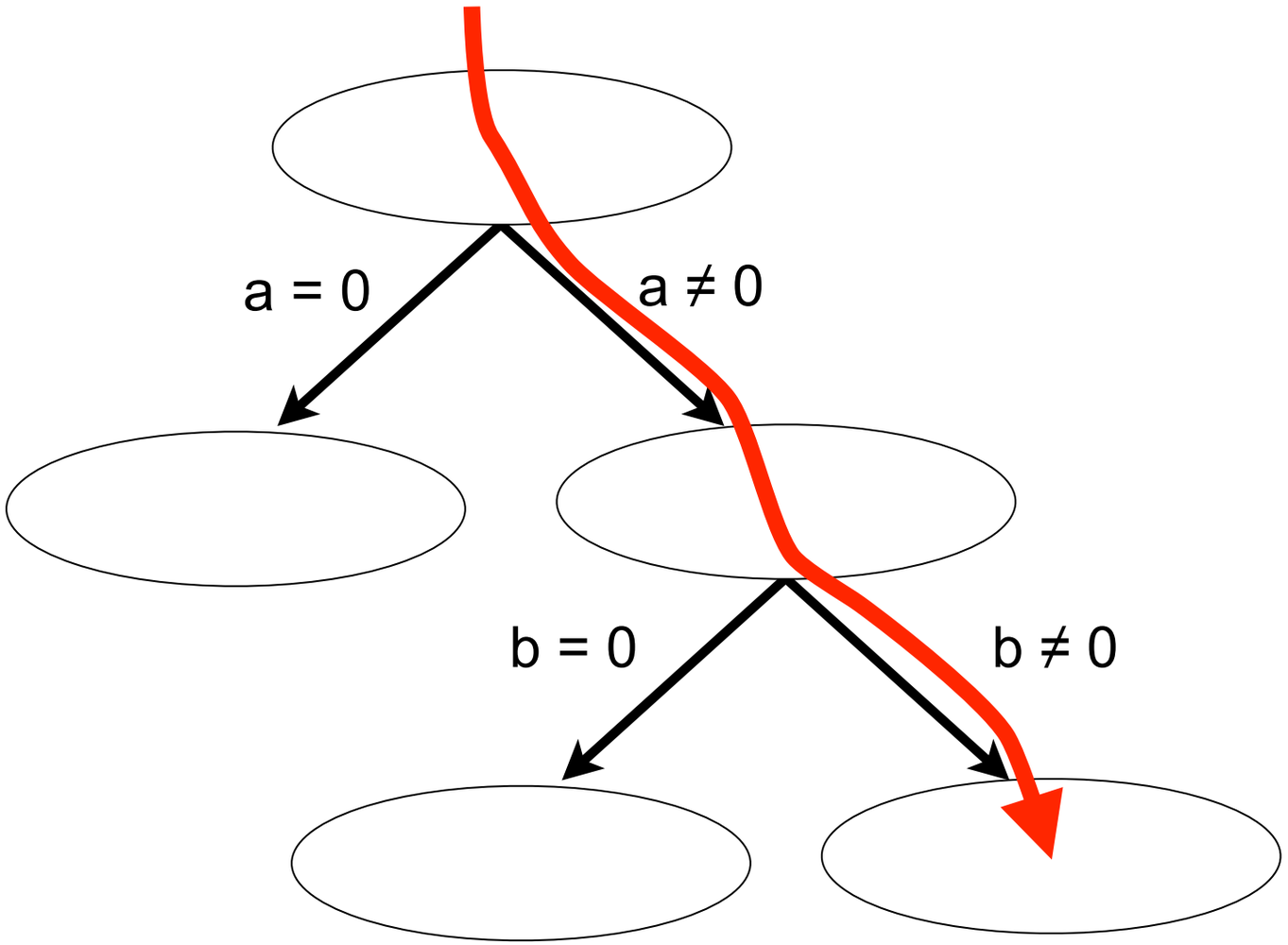


Infeasible paths



Infeasible paths

NICTA



Pruning infeasible paths

NICTA

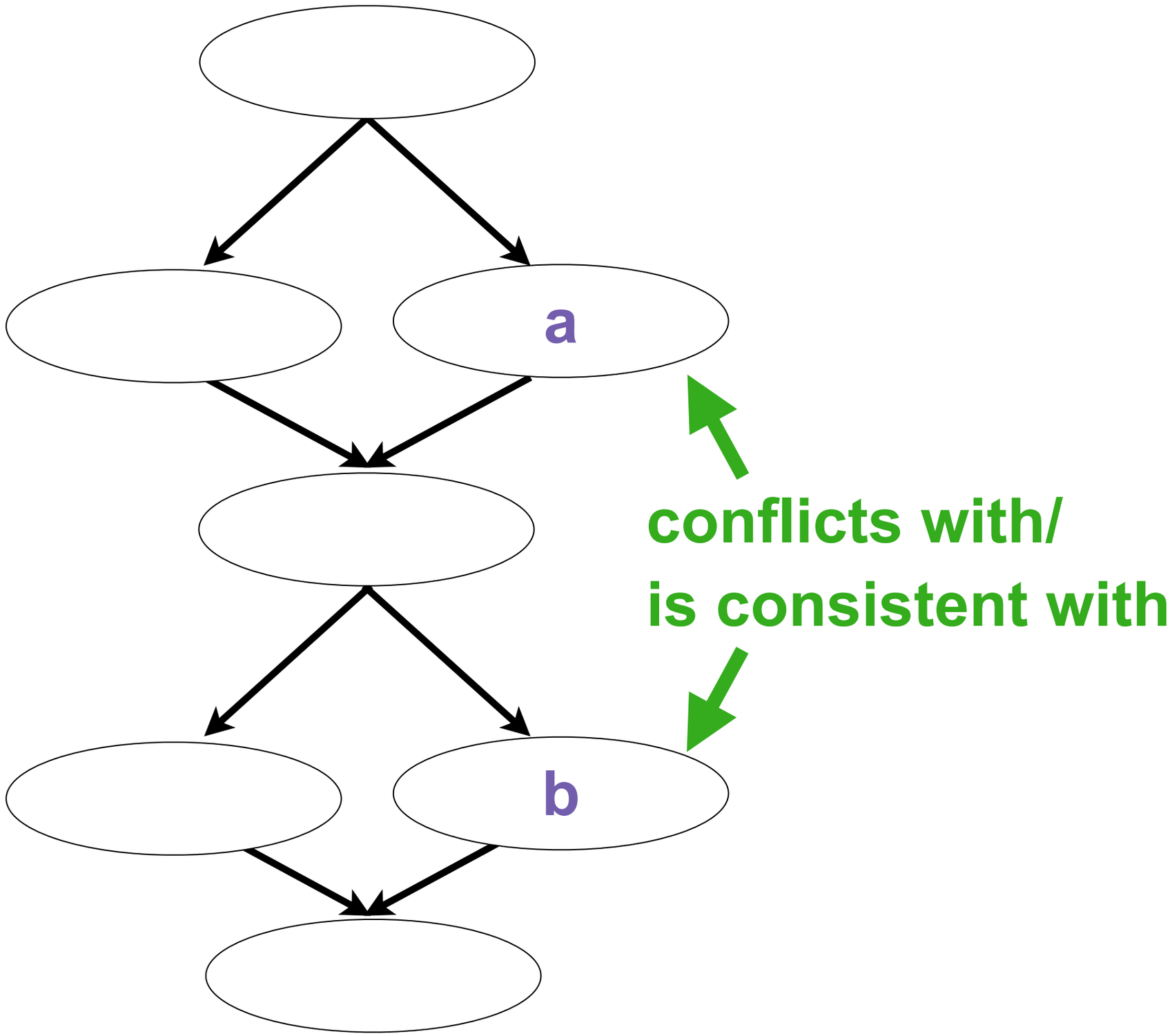
Express constraints as one of:

a **conflicts with** ***b*** when called under ***f***

or

a **is consistent with** ***b*** when called under ***f***

Infeasible paths



Verifying annotations

NICTA

- How can we verify these manual annotations?
- For two basic blocks **a, b**:
  - show all paths execute **a** and **b** the same number of times
  - show all paths execute at most one of **a** or **b**
- For some loop ℓ in the binary:
  - How many times can ℓ iterate?

Why model checking?

NICTA

- Many infeasible paths and even some loop bounds cannot be determined without program invariants

- Invariants are known to the formal proof

- Formal proof statements and invariants are a natural fit for model checkers

Why model checking?

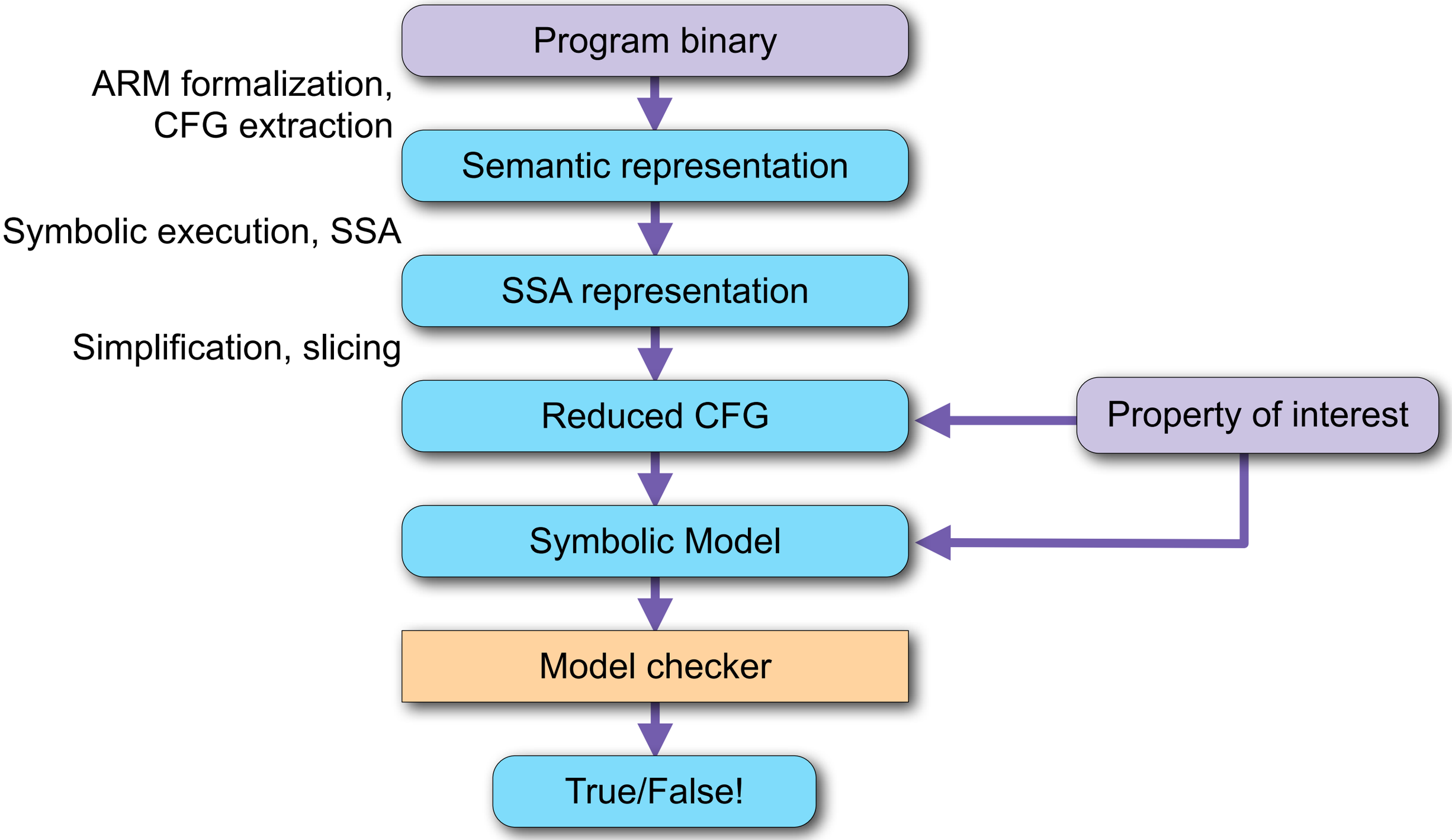
NICTA

```
int count_bits(uint32_t x) {
    int c = 0;
    while (x != 0) {
        x &= x - 1; /* clear lowest bit */
        c++;
    }
    return c;
}
```

What loop bound?

From binary → model checker

NICTA



Extracting instruction semantics

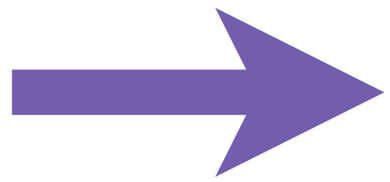
NICTA

- Reused existing formalization of ARM ISA

(Fox & Myreen, ITP 2010)

machine code: `E2813002`

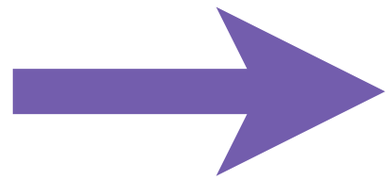
```
add r3, r1, #2
```



```
r3 ← r1 + 2
r15 ← r15 + 4
```

machine code: `E8AD0028`

```
stmia sp!, {r3, r5}
```

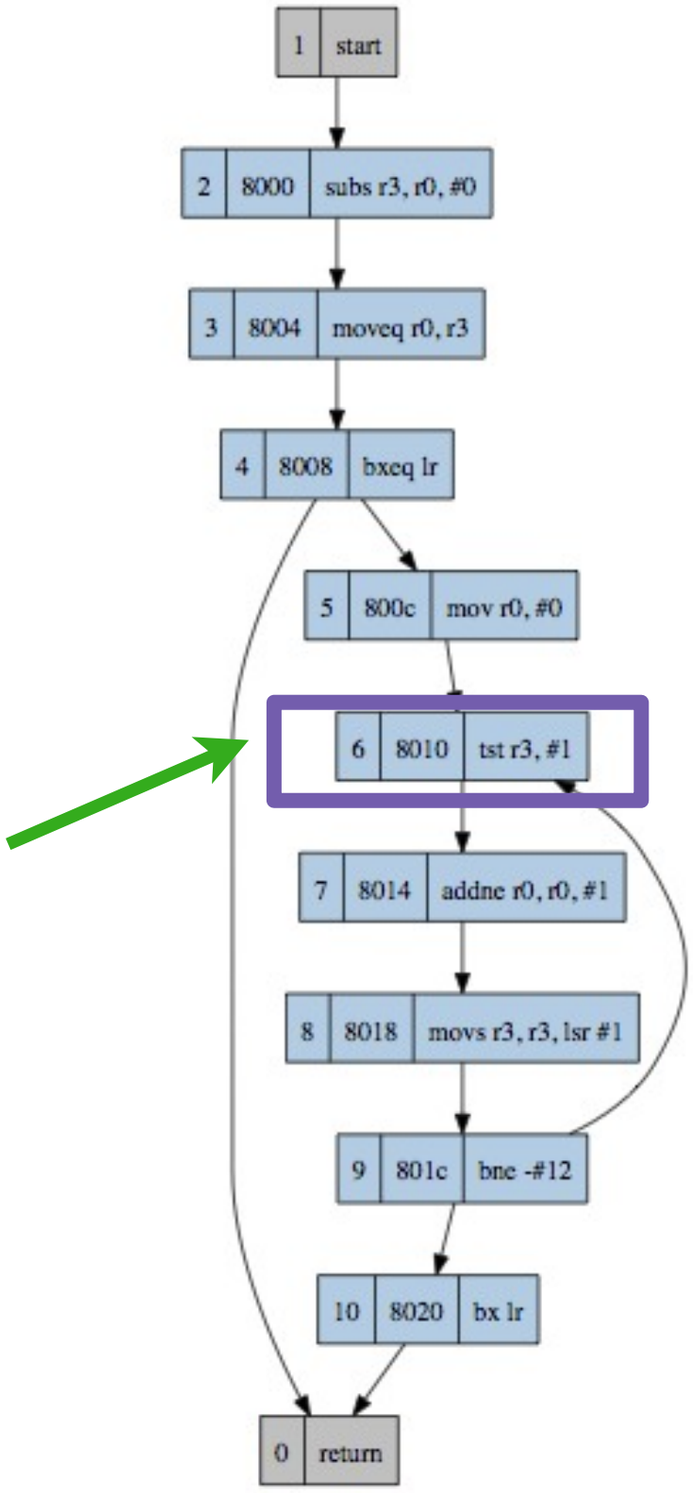


```
mem r13 ← r3<7:0>
mem (r13 + 1) ← r3<15:8>
mem (r13 + 2) ← r3<23:16>
mem (r13 + 3) ← r3<31:24>
mem (r13 + 4) ← r5<7:0>
mem (r13 + 5) ← r5<15:8>
mem (r13 + 6) ← r5<23:16>
mem (r13 + 7) ← r5<31:24>
r13 ← r13 + 8
r15 ← r15 + 4
```

Identifying properties of interest

NICTA

How many times can this node execute?

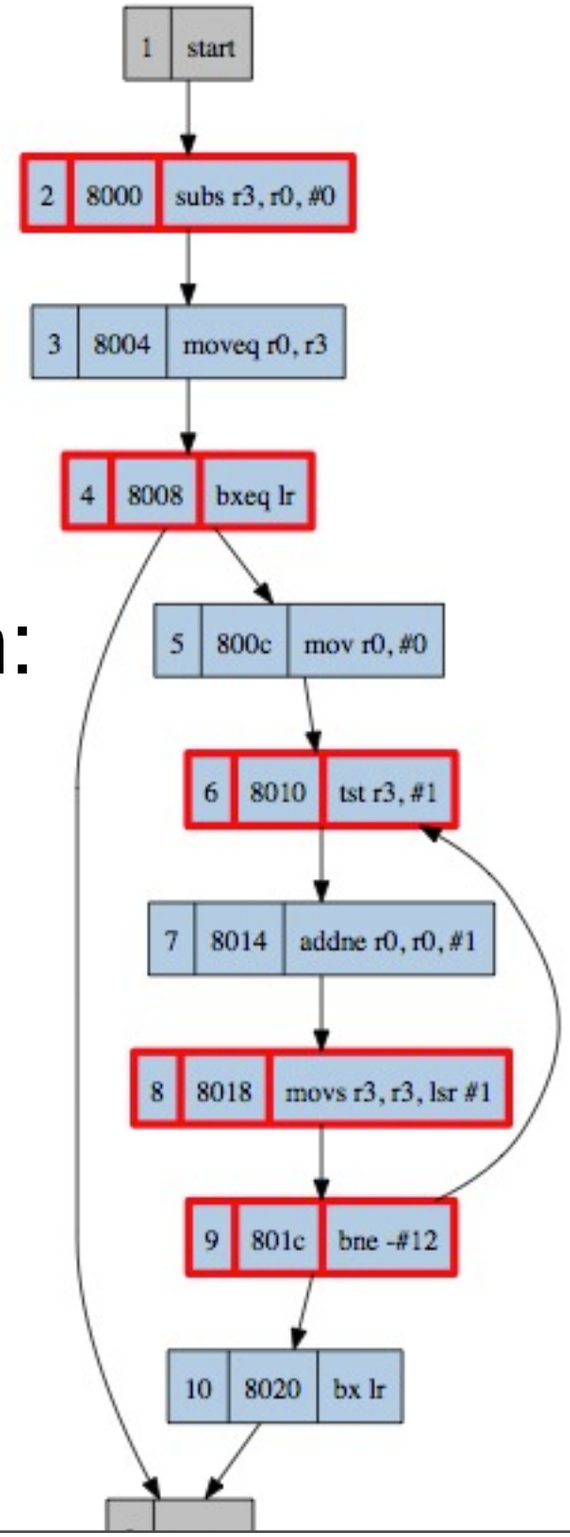


Computing a program slice

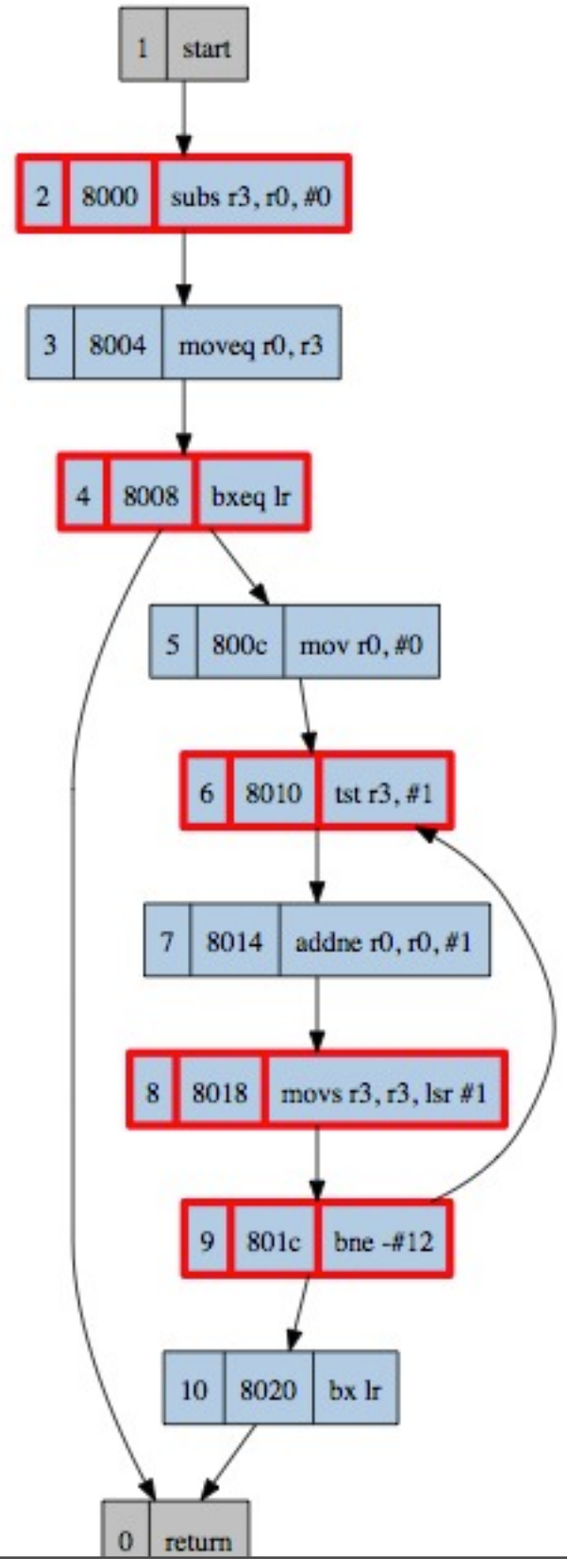
NICTA

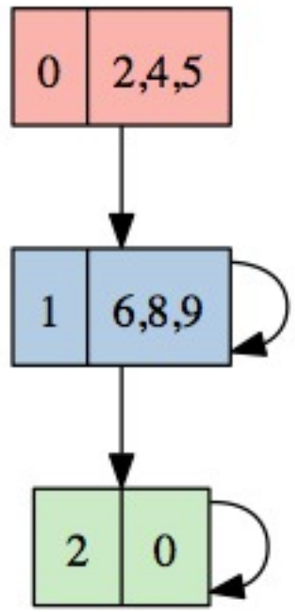
Find all nodes which can impact upon execution of that block through:

- data dependencies
- control dependencies



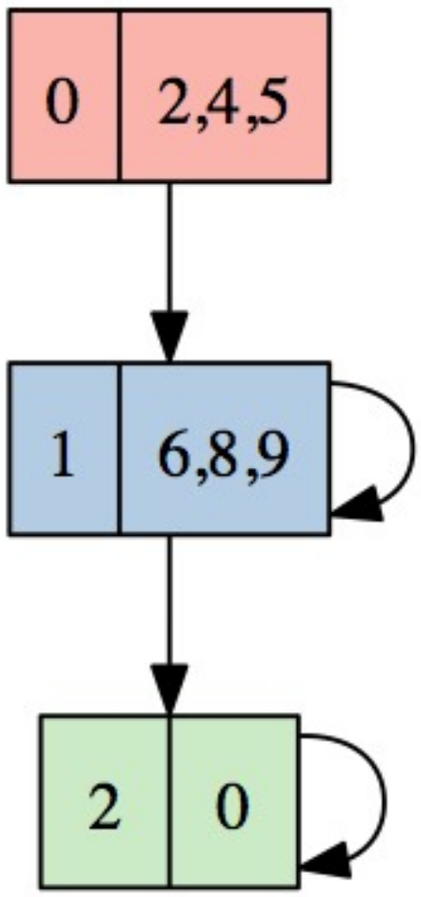
Reducing the CFG further





Model checking the CFG

NICTA



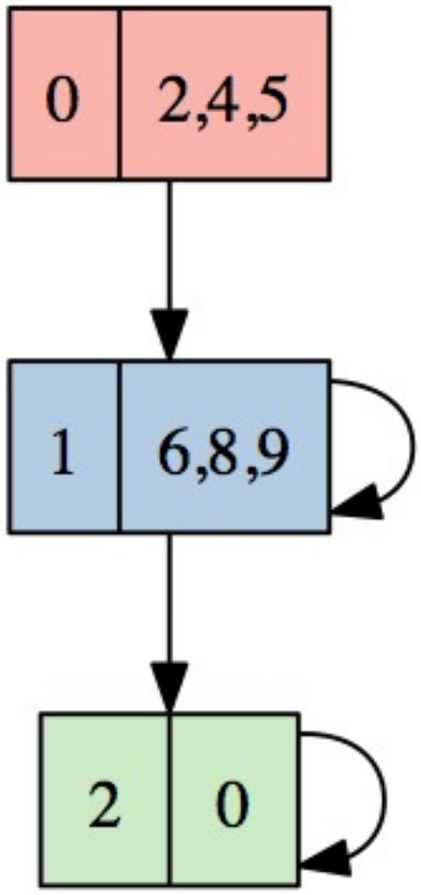

```
init(n) := 0
next(n) := case
  n=0: 1
  n=1 && cond_1: 1
  n=1 && !cond_1: 2
  n=2: 2
```

```
init(c) := 0
next(c) := case
  n=0: 0
  n=1: c + 1
  else: c
```

```
VAR
    n : -1..2;
    memRead : unsigned word[8];
    cond_3 : boolean;
    r3_2 : unsigned word [32];
    c : unsigned word[32];
DEFINE
    psrZ_3 := (((1) >= 32) ? 0ud32_
    cond_1 := !(psrZ_3);
    cond_2 := psrZ_1;
    r3_1 := r0_0;
    r3_3 := ((1) >= 32) ? 0ud32_0 :
    psrZ_1 := (r0_0) = (0ud32_0);
FROZENVAR
    r0_0 : unsigned word [32];
    r14_0 : unsigned word [32];
ASSIGN
    next(cond_3) := case
        n=1: cond_5;
        n=0: cond_2;
        TRUE: cond_3;
    esac;
    next(r3_2) := case
        n=1: r3_3;
        n=0: r3_1;
        TRUE: r3_2;
    esac;
```

Model checking the CFG

NICTA

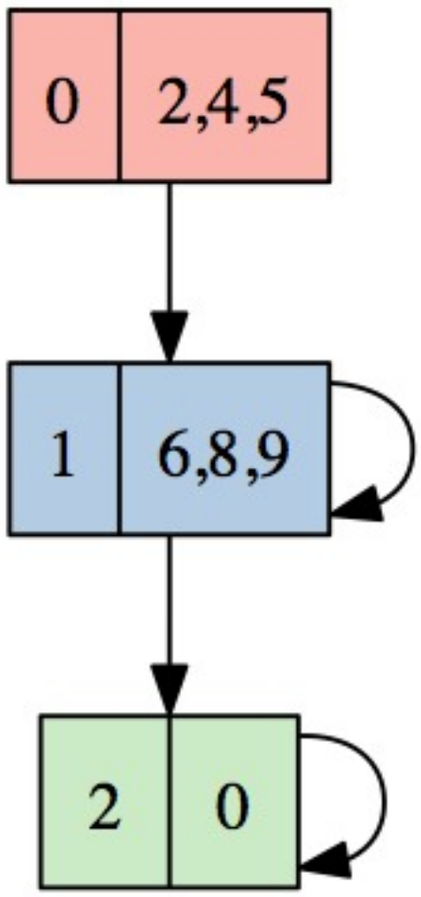


For loop counts:

Ask a model checker: is c < *k*?

(and binary search for *k*)

Model checking the CFG



For conflict constraints:

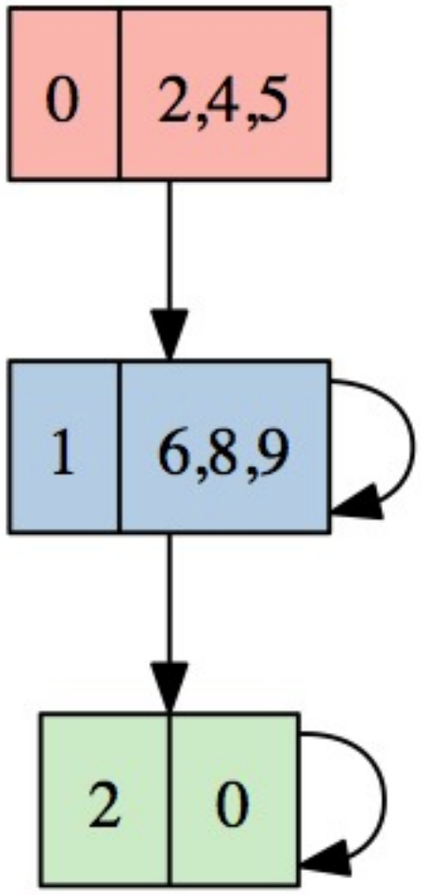
Add “visited” flag for nodes *a* and *b*

Ask a model checker to ensure that *a* and *b* are never both true

In CTL: `AG !(visitedA & visitedB)`

Model checking the CFG

NICTA



For consistency constraints:

Add “count” variables for nodes *a* and *b*

Ask a model checker to ensure counts for *a* and *b* are equal (eventually)

In CTL: `AF (countA = countB)`

Loop bound verification

NICTA

Can compute 18/32 loop bounds in seL4:

- 1 loop depends upon invariants in the proof
- 1 loop cannot be bounded due to complex exit conditions
  - model checker attempts to find the *smallest* loop bound
  - complex state space must be explored to deduce bound
- 12 loops, all identical in structure, cannot be bounded due to poor memory alias analysis

Loop bound verification

```
void f(uint32_t n)
{
  uint32_t i = 1 << n;

  if (i > 256)
    i = 256;

  while (i > 0) {
    ...
    i -= 4;
  }
}
```

If n ≥ 32, i is undefined

If n ≤ 1, loop is infinite

Infeasible paths

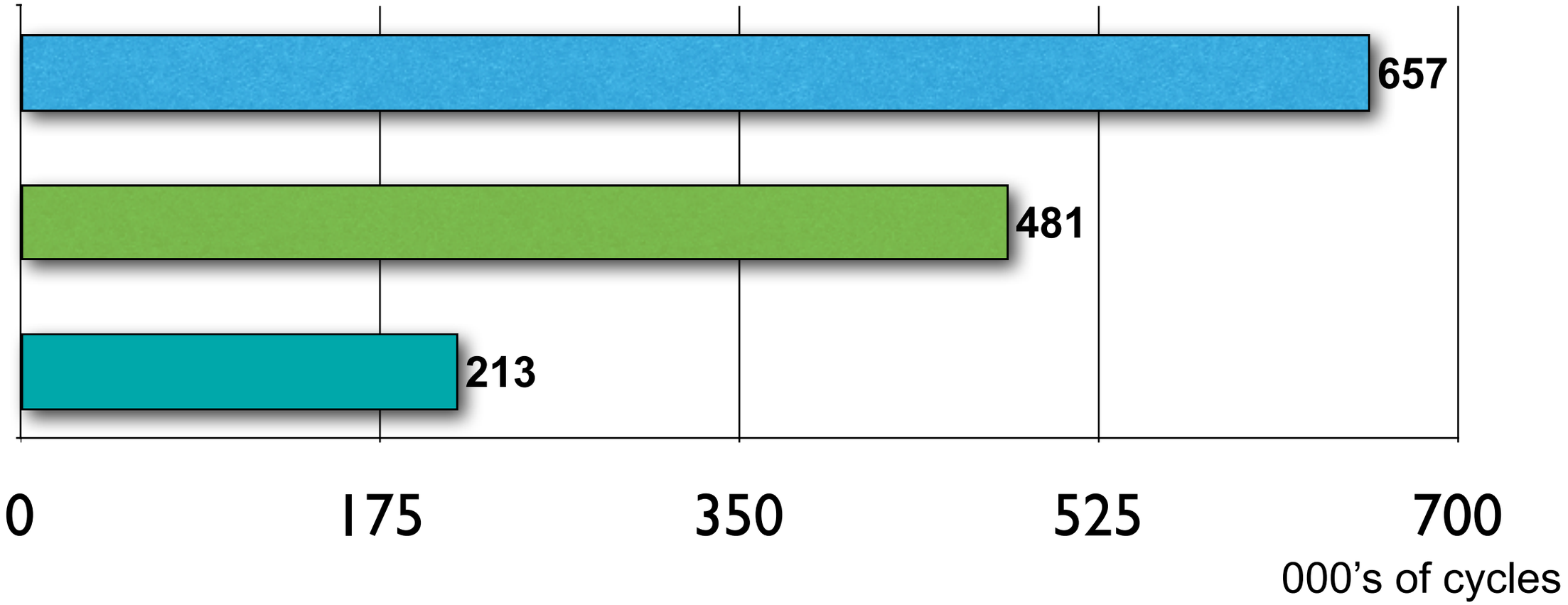
Of 35 infeasible path constraints

- 4 validated
- 1 shown untrue (oops!)
- 11 cannot be validated without better alias analysis
- 19 depend on kernel invariants

Results

NICTA

Estimated worst-case execution time of seL4



- Without infeasible path information
- With verified infeasible path information
- With all infeasible path information

Research directions

- Integrate with proof invariants
- Automate infeasible path detection (WIP)
- Use a faster ISA formalization

Summary

NICTA

Sequoll is able to:

- apply model checkers to reason about compiled ARM binaries
- validate manual infeasible path annotations
- compute “interesting” loop bounds
- (eventually) integrate formal proof with infeasible path information

➡ reduce scope for errors in WCET analysis!

Download it!

http://www.ssrg.nicta.com.au/software/TS/wcet-tools

Bernard.Blackham@nicta.com.au