Timing Analysis Enhancement for Synchronous Program

Pascal Raymond, Claire Maiza,

Catherine Parent-Vigouroux and Fabienne Carrier

Verimag/Grenoble-Alpes University

RTNS'13 - Sophia-Antipolis

This work is supported by the French ANR project W-SEPT

Design and validation of real-time (embedded) systems

Two complementary domains

- Synchronous Programming
  - ↪ rigorous design/programming methods
  - ↪ focuses on functionality *the system computes right*
- Timing analysis (here, static WCET estimation)
  - ↪ performed at *binary* level (relevance)
  - ↪ focuses on real-time *the system computes fast enough*

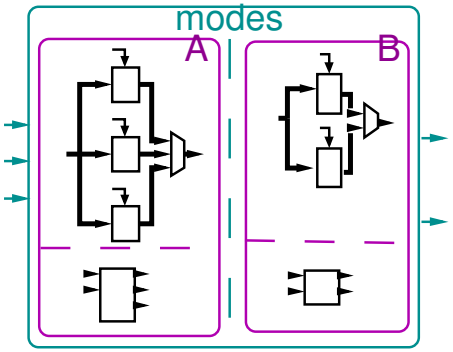
The Goal

Benefits from synchronous program semantics

to refute infeasible executions at the binary level,

and thus, possibly, enhance the WCET estimation

Synchronous Programming Workflow

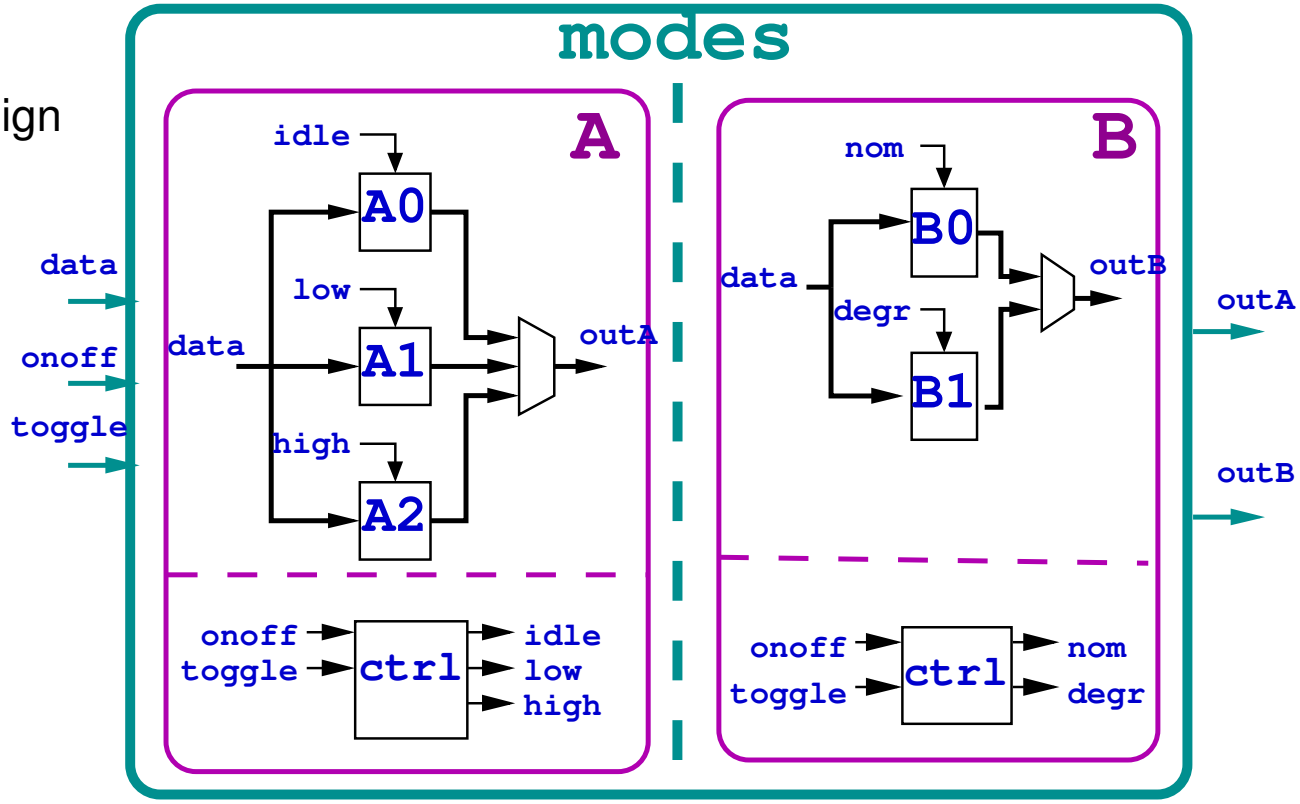


- Design level:
  - ↪ Concurrent, Hierarchic design
  - ↪ Idealized Concurrency
  - ↪ Behavior =

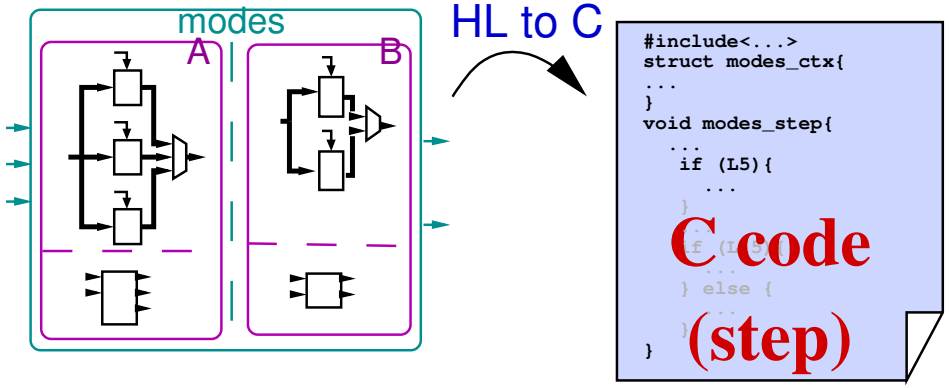
    sequence of reactions

    logical discrete time
  - ↪ Several styles/languages

    Here: data-flow/Lustre



Synchronous Programming Workflow

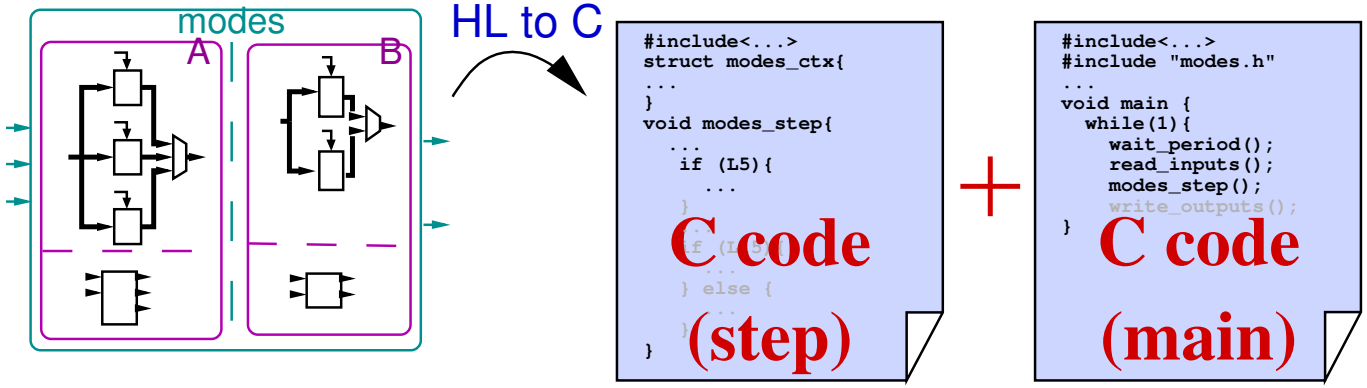


- Synchronous Compiler
  - ↪ Target language = C
  - ↪ Generates the step procedure
    (+ the necessary memory/ctx)
  - ↪ Basically: no more concurrency
    (static scheduling)
  - ↪ Simple sequential code
    here: no loops (*)
    DAG of (nested) if-then-else

(*) in general: static arrays/bounded “for” loops

```
#include<...>
struct modes_ctx{
...
}
void modes_step{
  ...
   if (L5){
     ...
   }
   ...
   if (L15){
     ...
   } else {
     ...
   }
}
```

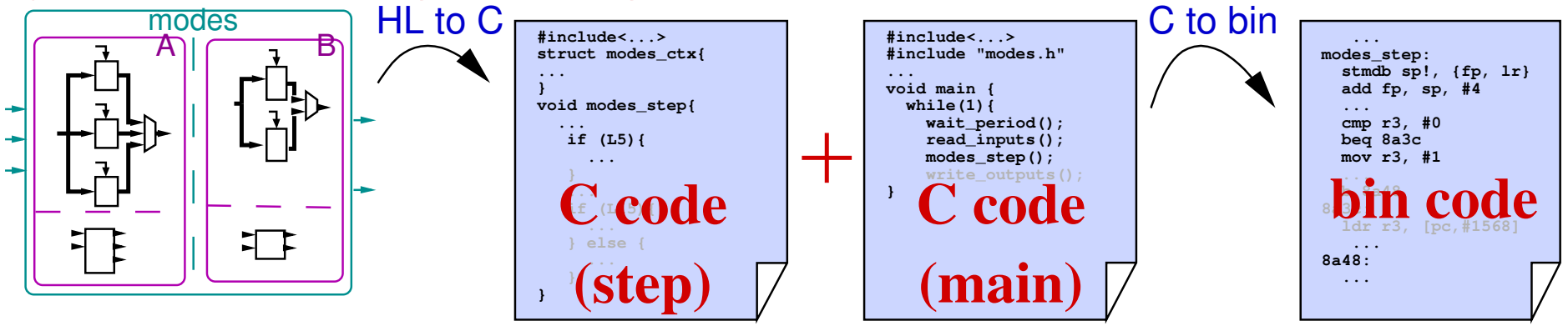
Synchronous Programming Workflow



- Example of main code
  - ↪ Basically an infinite loop
  - ↪ Each loop performs one reaction
  - ↪ Depends on system choices
    periodic/event-driven etc.

```
#include<...>
#include "modes.h"
...
void main {
  while(1){
    wait_period();
    read_inputs();
    modes_step();
    write_outputs();
}
```

Synchronous Programming Workflow



- Binary code
  - ↪ via arm-elf-gcc
  - ↪ WCET estimation should be done here

  for **`modes_step`**
  i.e. a step of main infinite loop

```
    ...
modes_step:
  stmdb sp!, {fp, lr}
  add fp, sp, #4
  ...
  cmp r3, #0
  beq 8a3c
  mov r3, #1
  ...
  b 8a48
8a3c:
  ldr r3, [pc,#1568]
   ...
8a48:
  ...
```

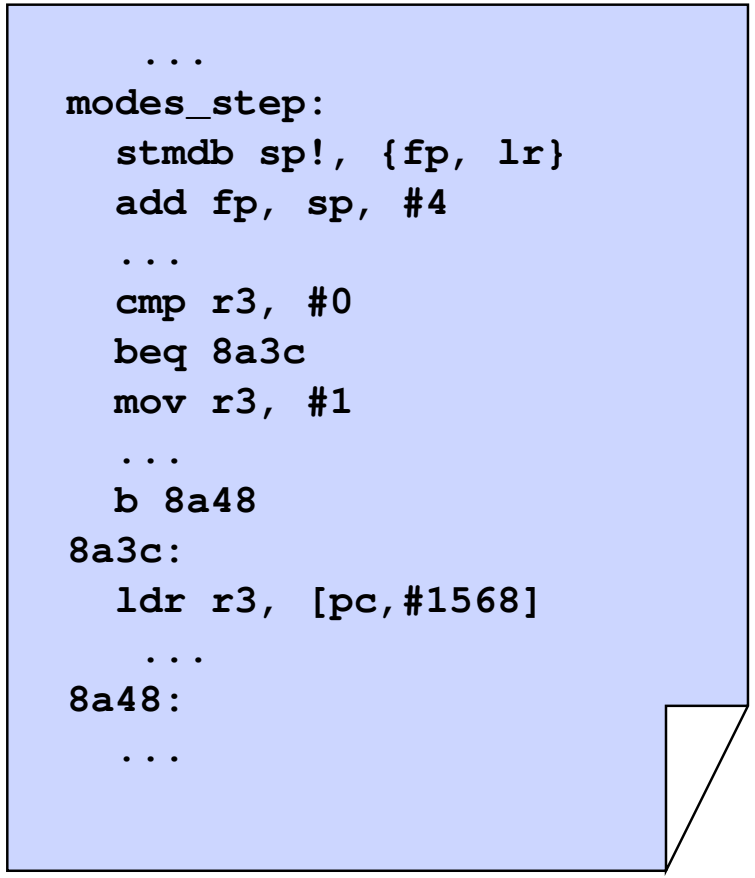
WCET Estimation Workflow

- Here: OTAWA

```
    ...
modes_step:
  stmdb sp!, {fp, lr}
  add fp, sp, #4
  ...
  cmp r3, #0
  beq 8a3c
  mov r3, #1
  ...
  b 8a48
8a3c:
  ldr r3, [pc,#1568]
   ...
8a48:
  ...
```

WCET Estimation Workflow

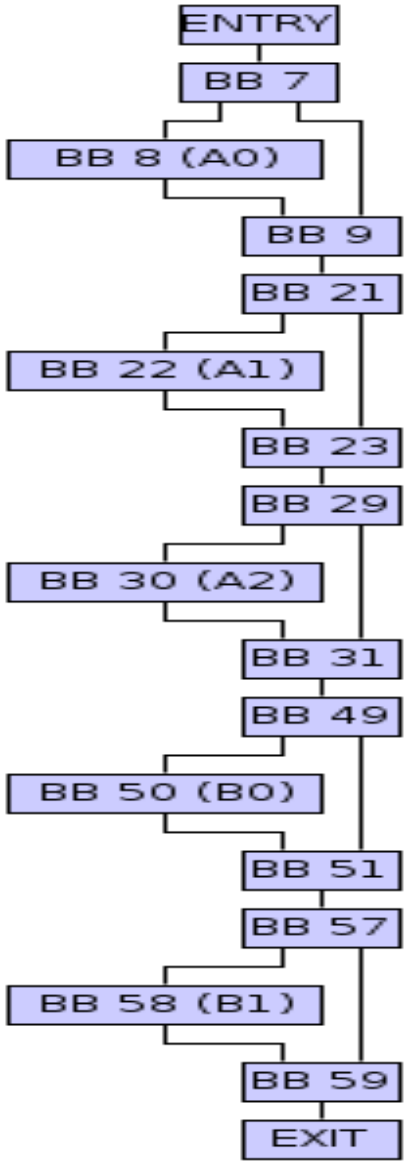
- Here: OTAWA
- Control Flow Graph (CFG) reconstruction



```
    ...
modes_step:
  stmdb sp!, {fp, lr}
  add fp, sp, #4
  ...
  cmp r3, #0
  beq 8a3c
  mov r3, #1
  ...
  b 8a48
8a3c:
  ldr r3, [pc,#1568]
   ...
8a48:
  ...
```

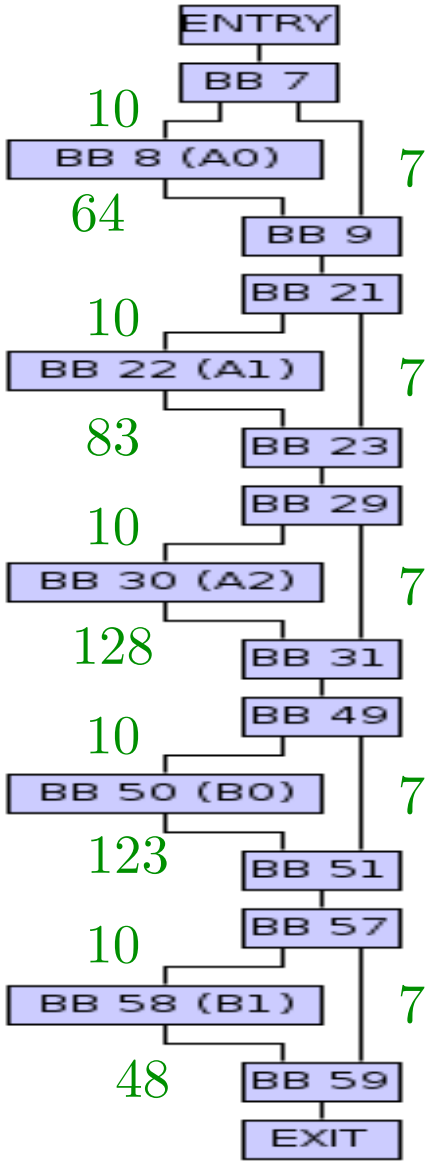
WCET Estimation Workflow

- Here: OTAWA
- Control Flow Graph (CFG) reconstruction
  ↪ Basic Blocks + edges (small part here)



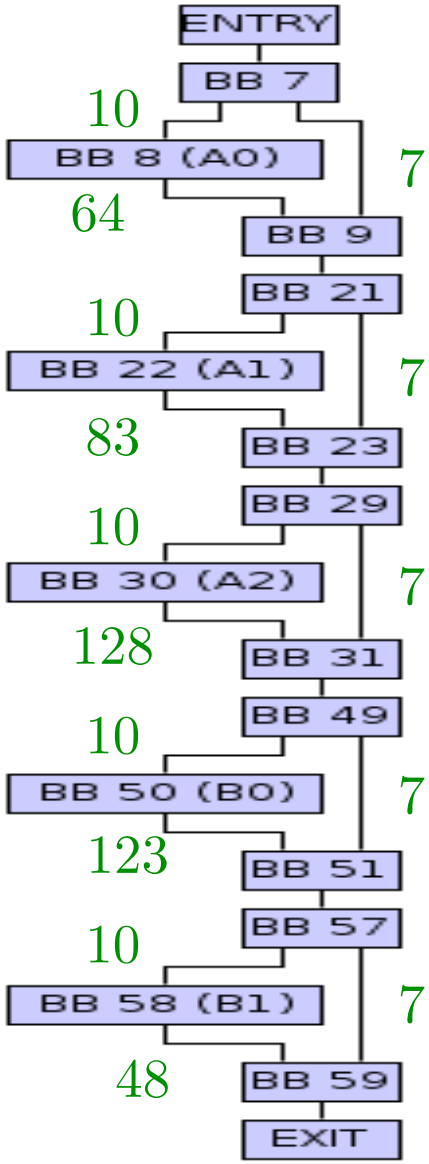
WCET Estimation Workflow

- Here: OTAWA
- Control Flow Graph (CFG) reconstruction
  ↪ Basic Blocks + edges (small part here)
- μ-archi analysis
  ↪ local costs, $c_{i,j}$, in cpu cycles



WCET Estimation Workflow

- Here: OTAWA
- Control Flow Graph (CFG) reconstruction
  ↪ Basic Blocks + edges (small part here)
- μ-archi analysis
  ↪ local costs, $c_{i,j}$, in cpu cycles
- *Data-flow analysis*
  ↪ *loop bounds + others (not here)*

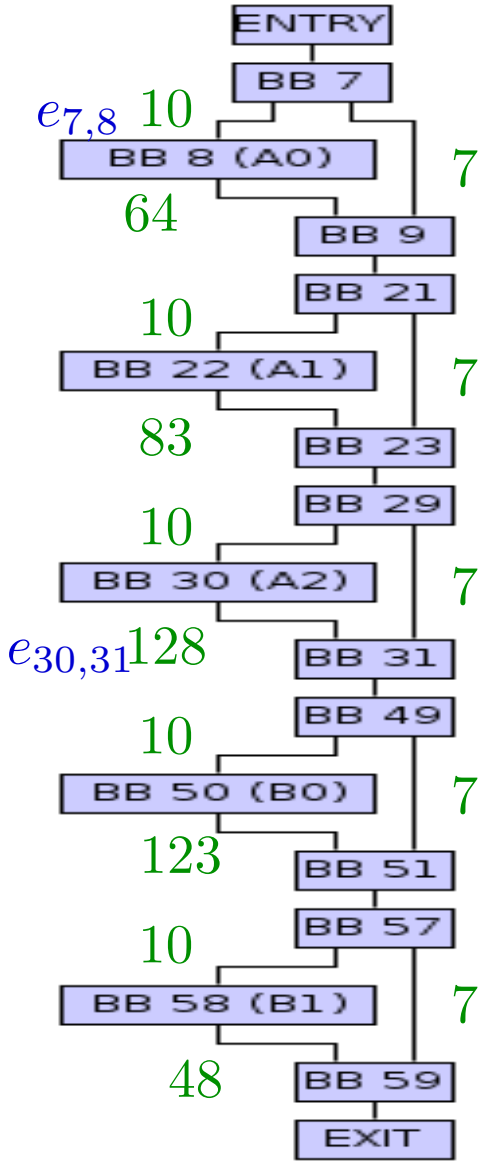


WCET Estimation Workflow

- Here: OTAWA
- Control Flow Graph (CFG) reconstruction
  ↪ Basic Blocks + edges (small part here)
- μ-archi analysis
  ↪ local costs, $c_{i,j}$, in cpu cycles
- *Data-flow analysis*
  ↪ *loop bounds + others (not here)*
- Implicit Path Enumeration Technique (IPET)

  Integer Linear Programming encoding

  ↪ one counter variable per edge ($e_{i,j}$)
  (n.b. here, $e_{i,j}$ = 0 or 1)

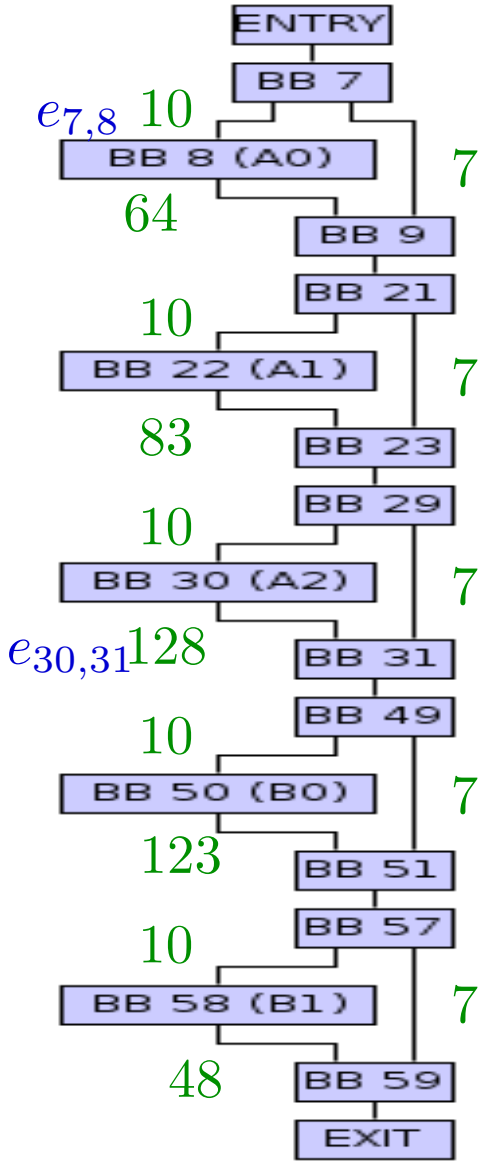


WCET Estimation Workflow

- Here: OTAWA
- Control Flow Graph (CFG) reconstruction
  ↪ Basic Blocks + edges (small part here)
- μ-archi analysis
  ↪ local costs, $c_{i,j}$, in cpu cycles
- *Data-flow analysis*
  ↪ *loop bounds + others (not here)*
- Implicit Path Enumeration Technique (IPET)

  Integer Linear Programming encoding

  ↪ one counter variable per edge ($e_{i,j}$)
  (n.b. here, $e_{i,j}$ = 0 or 1)
  ↪ Structural Constraints: $\Sigma e_{i,j} = \Sigma e_{j,k}$
  (and indeed: entry = exit = 1)



WCET Estimation Workflow

- Here: OTAWA
- Control Flow Graph (CFG) reconstruction
  ↪ Basic Blocks + edges (small part here)
- μ-archi analysis
  ↪ local costs, $c_{i,j}$, in cpu cycles
- *Data-flow analysis*
  ↪ *loop bounds + others (not here)*
- Implicit Path Enumeration Technique (IPET)

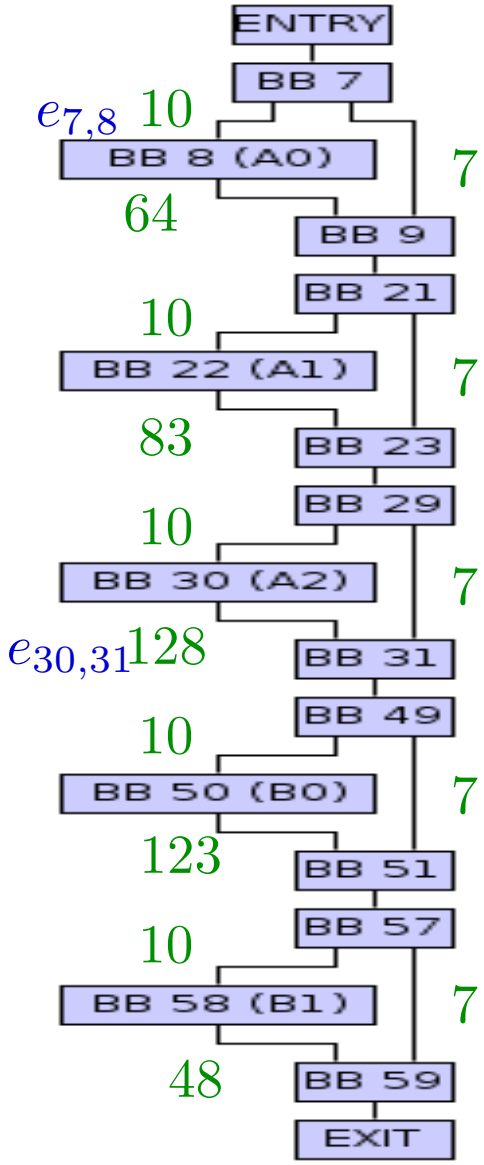
  Integer Linear Programming encoding

  ↪ one counter variable per edge ($e_{i,j}$)
  (n.b. here, $e_{i,j}$ = 0 or 1)
  ↪ Structural Constraints: $\Sigma e_{i,j} = \Sigma e_{j,k}$

  (and indeed: entry = exit = 1)

  ↪ Semantics Constraints

  loop bounds (not here), others ?

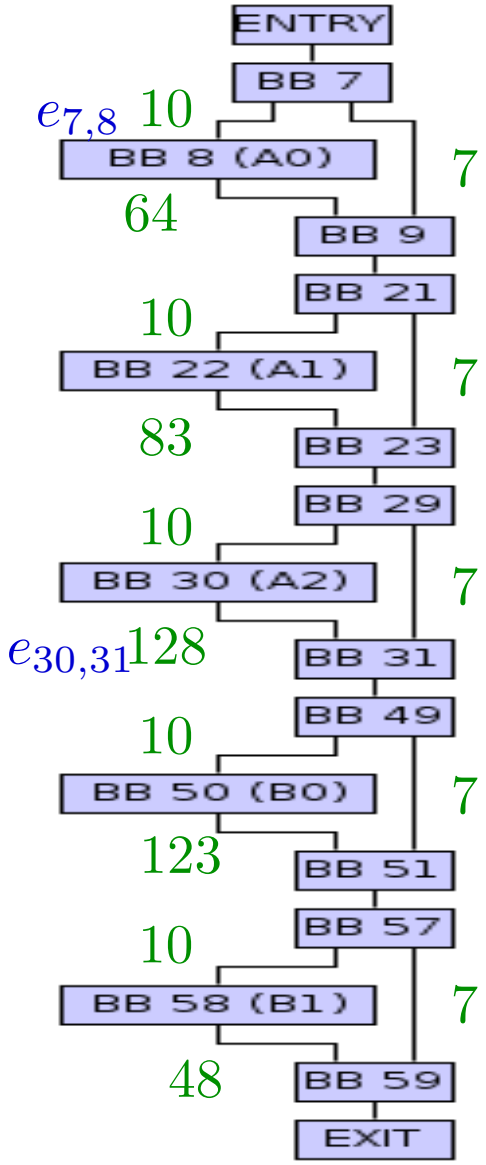


WCET Estimation Workflow

- Here: OTAWA
- Control Flow Graph (CFG) reconstruction
  ↪ Basic Blocks + edges (small part here)
- μ-archi analysis
  ↪ local costs, $c_{i,j}$, in cpu cycles
- *Data-flow analysis*
  ↪ *loop bounds + others (not here)*
- Implicit Path Enumeration Technique (IPET)

  Integer Linear Programming encoding

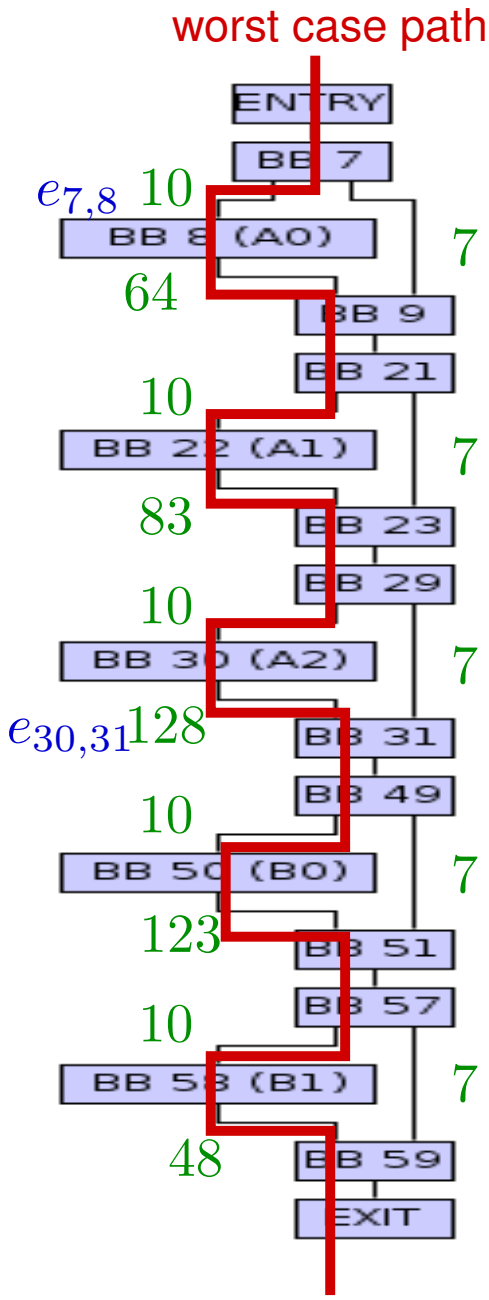
  ↪ one counter variable per edge ($e_{i,j}$)
  (n.b. here, $e_{i,j}$ = 0 or 1)
  ↪ Structural Constraints: $\Sigma e_{i,j} = \Sigma e_{j,k}$
  (and indeed: entry = exit = 1)
  ↪ Semantics Constraints
  loop bounds (not here), others ?
  ↪ Objective: MAX $\Sigma c_{i,j} \times e_{i,j}$



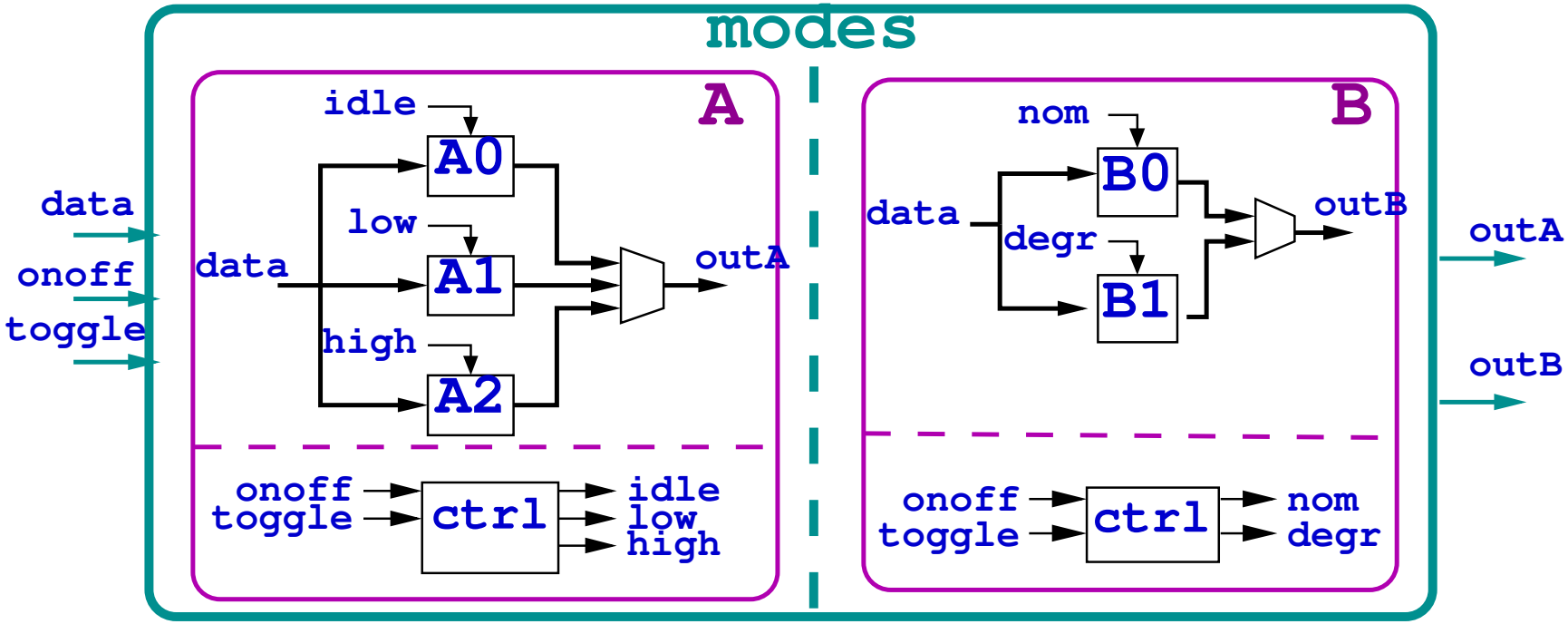
WCET Estimation Workflow

- Here: OTAWA
- Control Flow Graph (CFG) reconstruction
  ↪ Basic Blocks + edges (small part here)
- μ-archi analysis
  ↪ local costs, $c_{i,j}$, in cpu cycles
- *Data-flow analysis*
  ↪ *loop bounds + others (not here)*
- Implicit Path Enumeration Technique (IPET)

  Integer Linear Programming encoding
  ↪ one counter variable per edge ($e_{i,j}$)
  (n.b. here, $e_{i,j}$ = 0 or 1)
  ↪ Structural Constraints: $\Sigma e_{i,j} = \Sigma e_{j,k}$
  (and indeed: entry = exit = 1)
  ↪ Semantics Constraints
  loop bounds (not here), others ?
  ↪ Objective: MAX $\Sigma c_{i,j} \times e_{i,j}$
- Call an ILP Solver (here LPSolve)
  ↪ get 496 + the left-most path

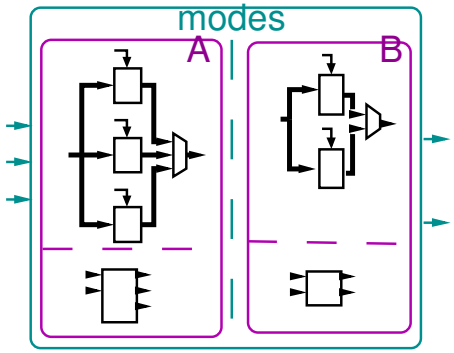


High Level Properties (that may help)



- Programming pattern: computation modes, based on clock-enable construct
- Intra-module exclusions: between **A0**, **A1**, **A2**, and between **B0** and **B1**
  may or may not be “obvious” on the code (i.e. structural)
- Inter-module exclusions: not in mode **A0** implies mode **B1**
  no chance to be obvious on the code
- In all cases: relatively “complex” properties
  infinite loop invariants, unlikely to be discovered by analysing C or bin code

Traceability problem

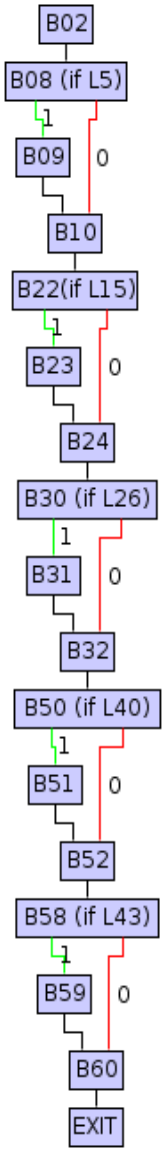


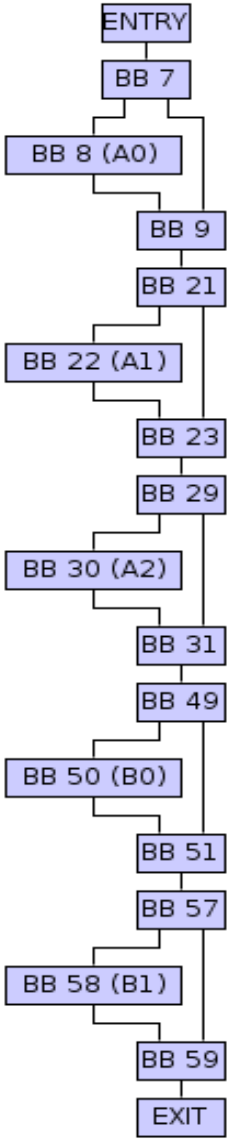
HL Compil
(Lustre → C)

C CFG

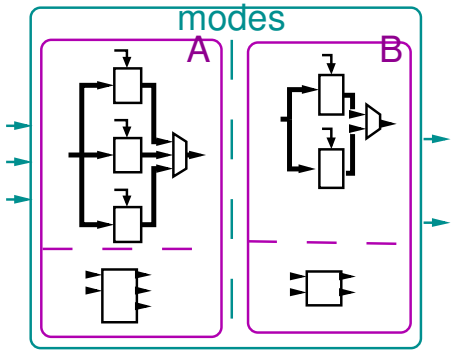
Code generation
(gcc)

bin CFG





Traceability problem

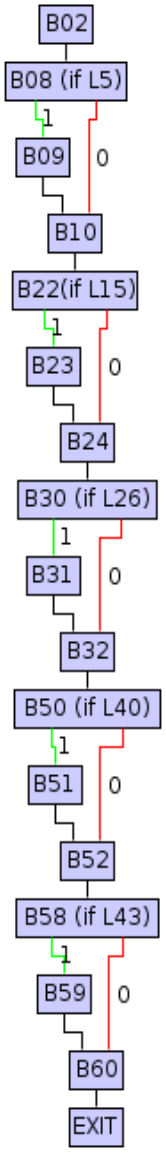


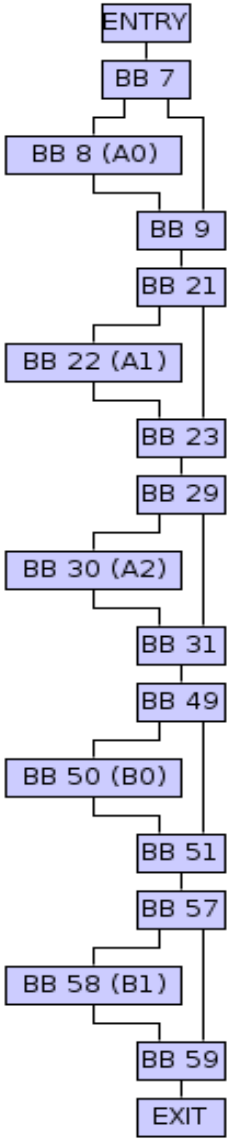
HL Compil
(Lustre → C)

C CFG

Code generation
(gcc)

bin CFG

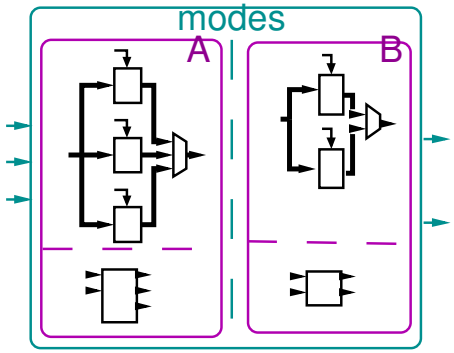




- problem: relate branches in bin CFG

to branches in C-CFG, and then predicates (variables) at the HL level

Traceability problem

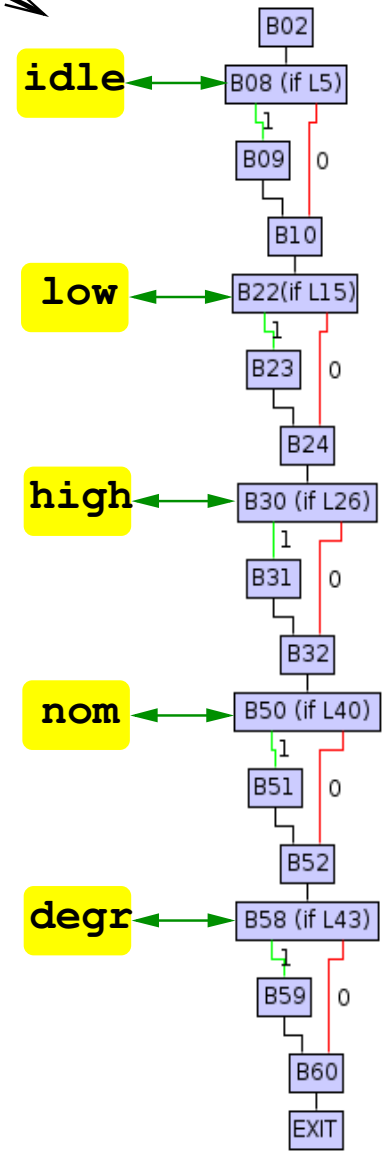


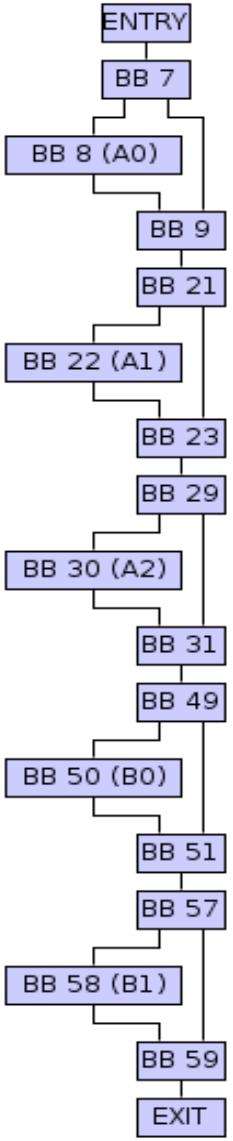
HL Compil
(Lustre → C)

C CFG

Code generation
(gcc)

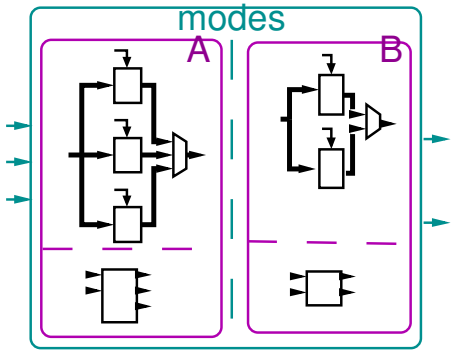
bin CFG





- problem: relate branches in bin CFG to branches in C-CFG, and then predicates (variables) at the HL level
- between HL and C: not a problem (compiler annotations)

Traceability problem

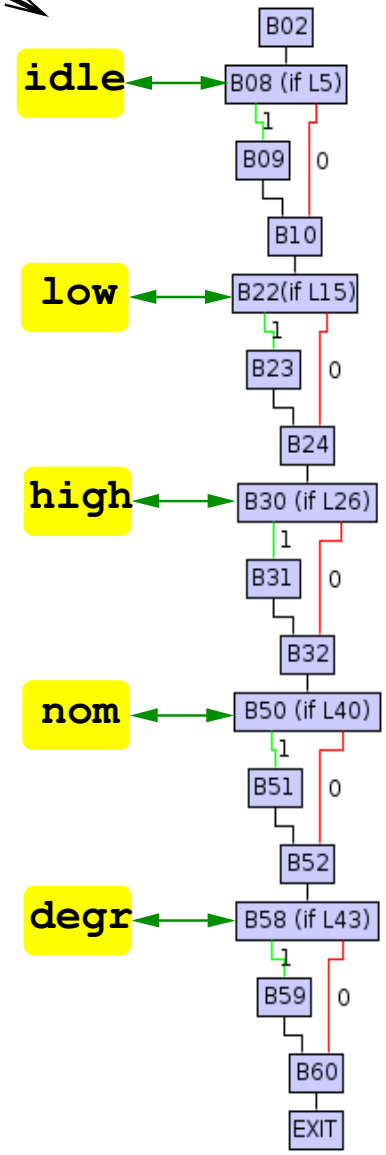


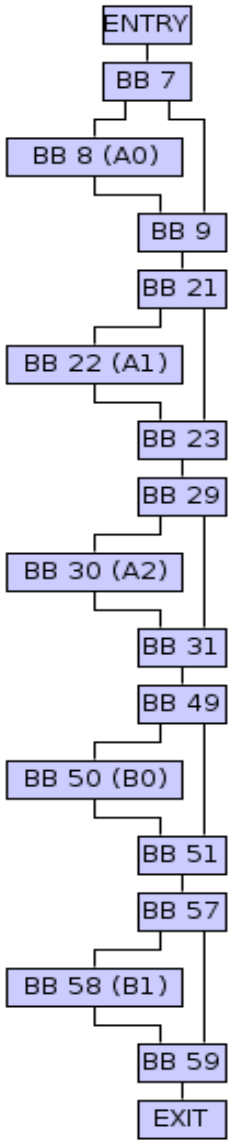
HL Compil
(Lustre → C)

C CFG

Code generation
(gcc)

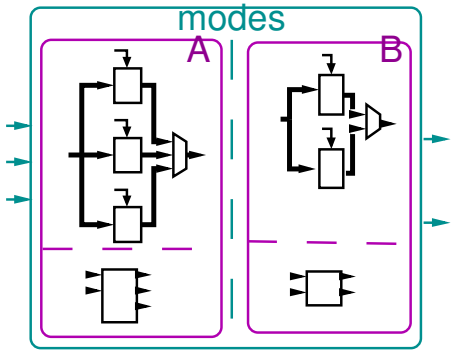
bin CFG





- problem: relate branches in bin CFG
  to branches in C-CFG, and then
  predicates (variables) at the HL level
- between HL and C: not a problem
  (compiler annotations)
- between C and bin: more difficult
  (simple heuristic: rely on debugging info)

Traceability problem

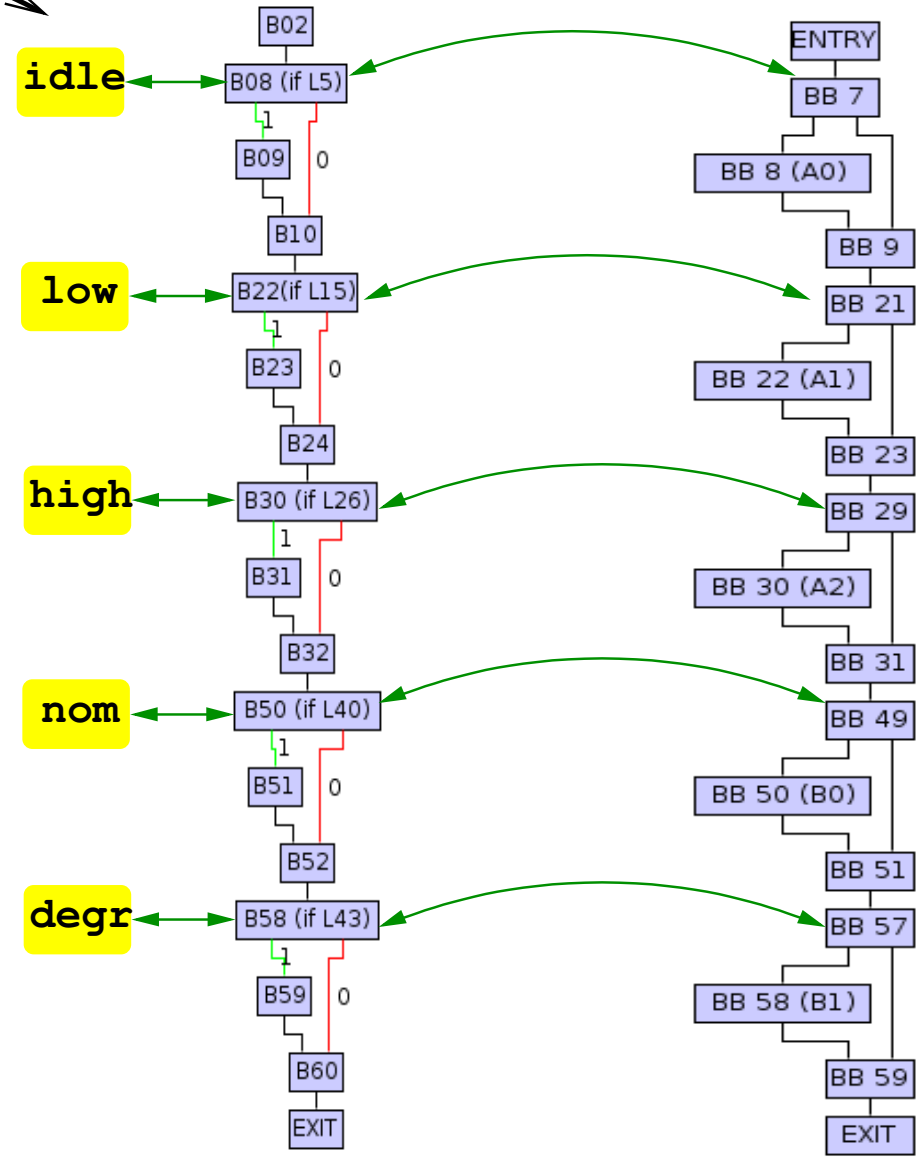


HL Compil
(Lustre → C)

Code generation
(gcc)

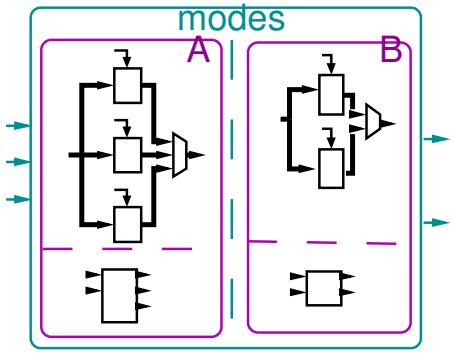
C CFG

bin CFG



- problem: relate branches in bin CFG
  to branches in C-CFG, and then
  predicates (variables) at the HL level
- between HL and C: not a problem
  (compiler annotations)
- between C and bin: more difficult
  (simple heuristic: rely on debugging info)
  ↪ No optimization (−O0)
  CFG's strictly match

Traceability problem

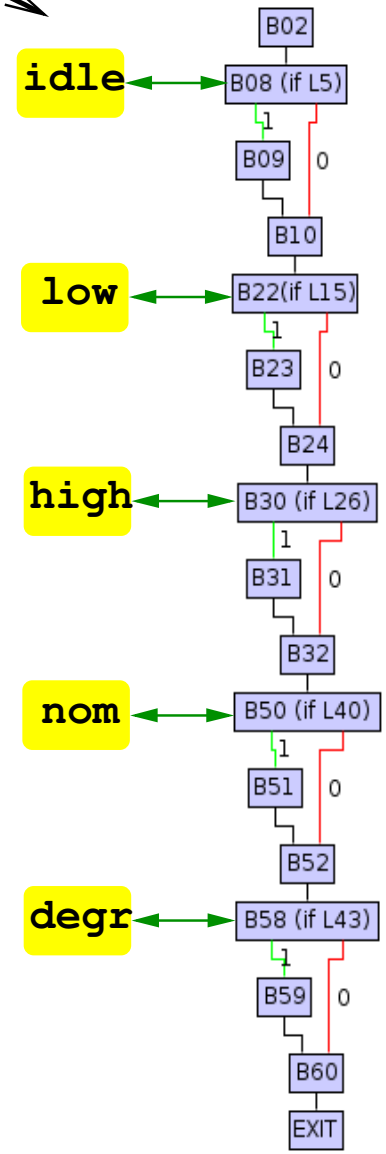


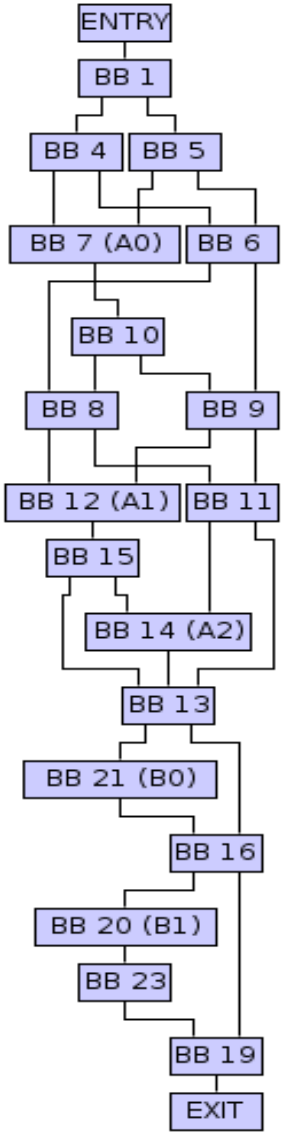
HL Compil
(Lustre → C)

C CFG

Code generation
(gcc)

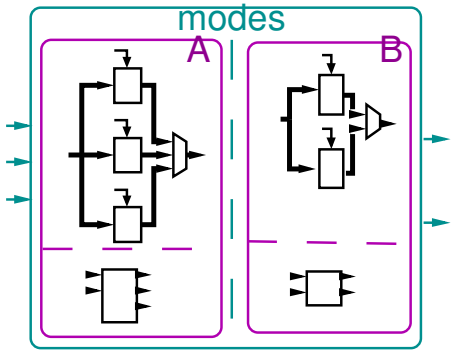
bin CFG





- problem: relate branches in bin CFG
  to branches in C-CFG, and then
  predicates (variables) at the HL level
- between HL and C: not a problem
  (compiler annotations)
- between C and bin: more difficult
  (simple heuristic: rely on debugging info)
  - ↪ No optimization (`-O0`)
    CFG's strictly match
  - ↪ Optimization (`-O2`)
    CFG relatively obfuscated

Traceability problem

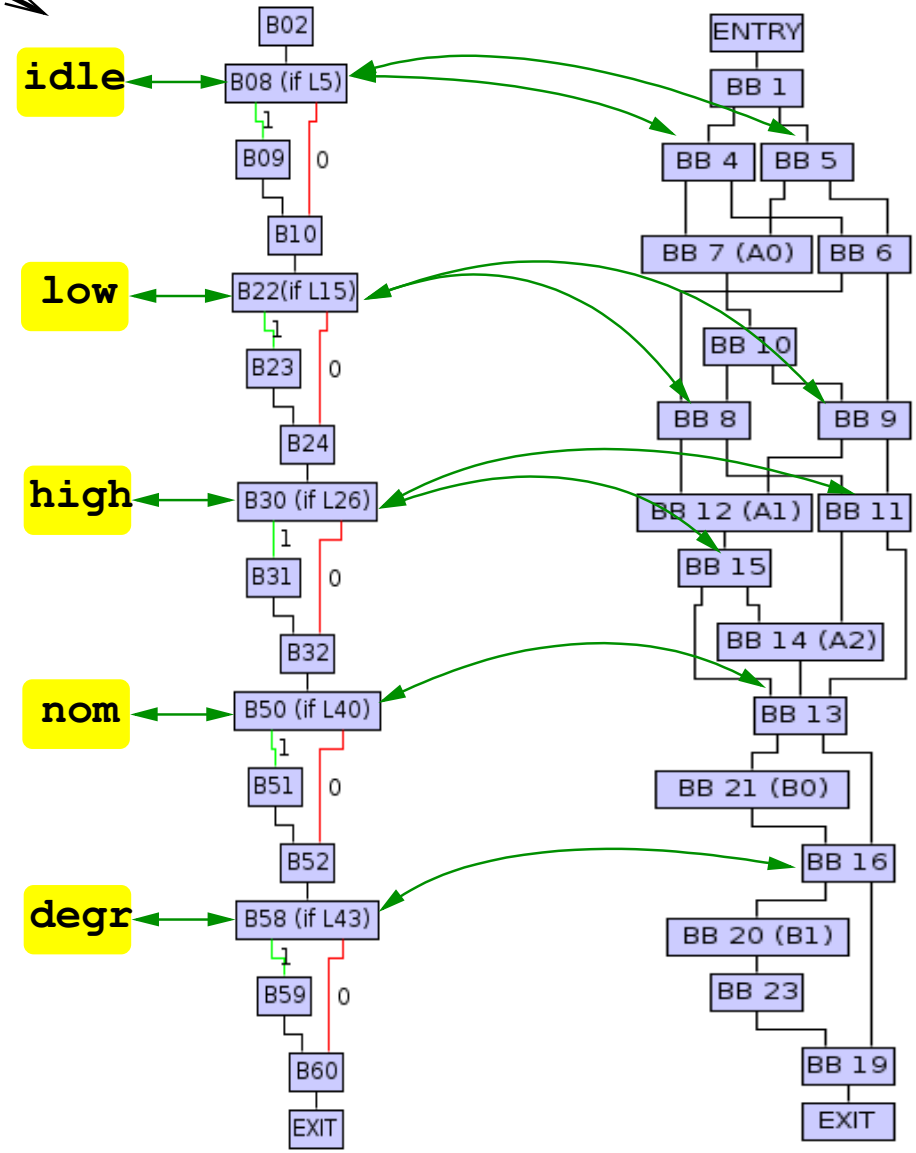


HL Compil
(Lustre → C)

Code generation
(gcc)

C CFG

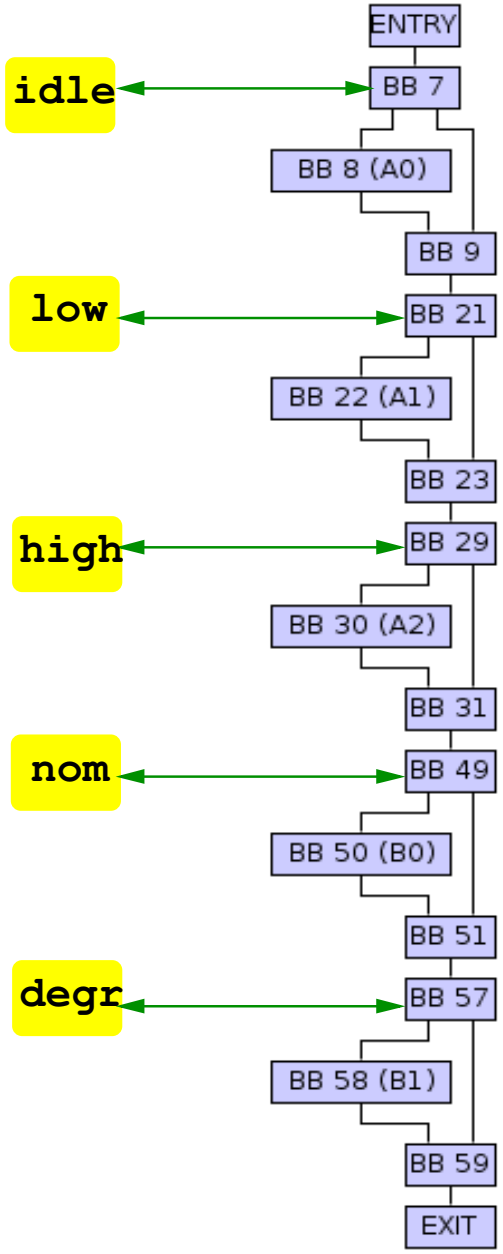
bin CFG



- problem: relate branches in bin CFG to branches in C-CFG, and then predicates (variables) at the HL level
- between HL and C: not a problem (compiler annotations)
- between C and bin: more difficult (simple heuristic: rely on debugging info)
  - ↪ No optimization (`-O0`)
    CFG's strictly match
  - ↪ Optimization (`-O2`)
    CFG relatively obfuscated
    but debug info still works

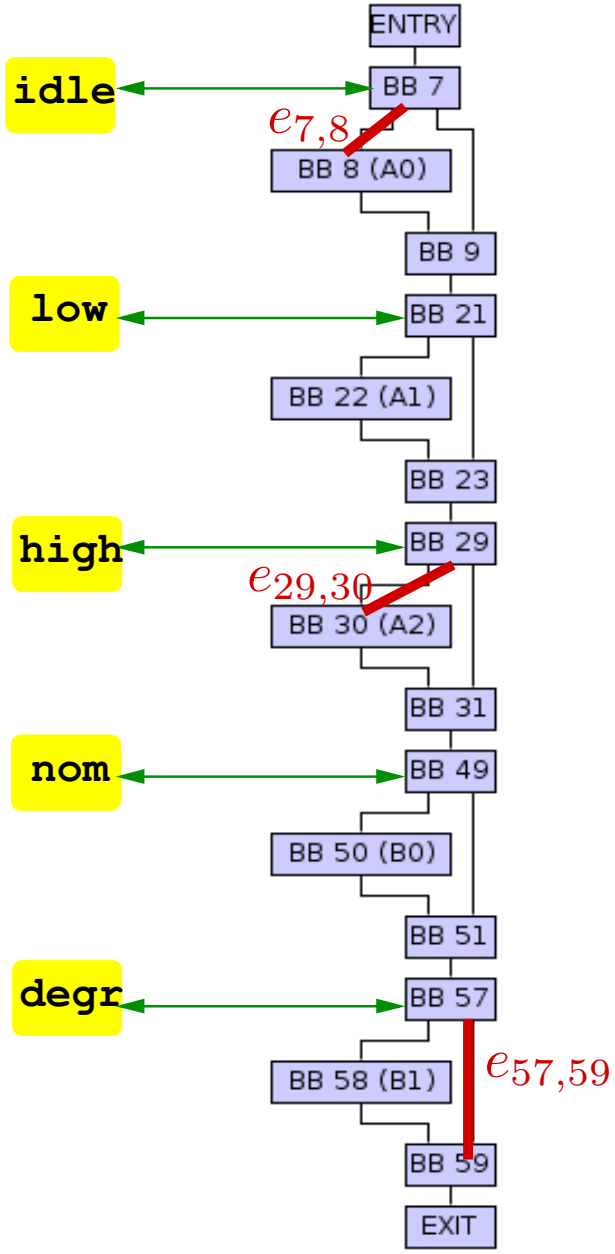
From (binary) paths to (HL) properties to (ILP) constraints

- Traceability has been achieved



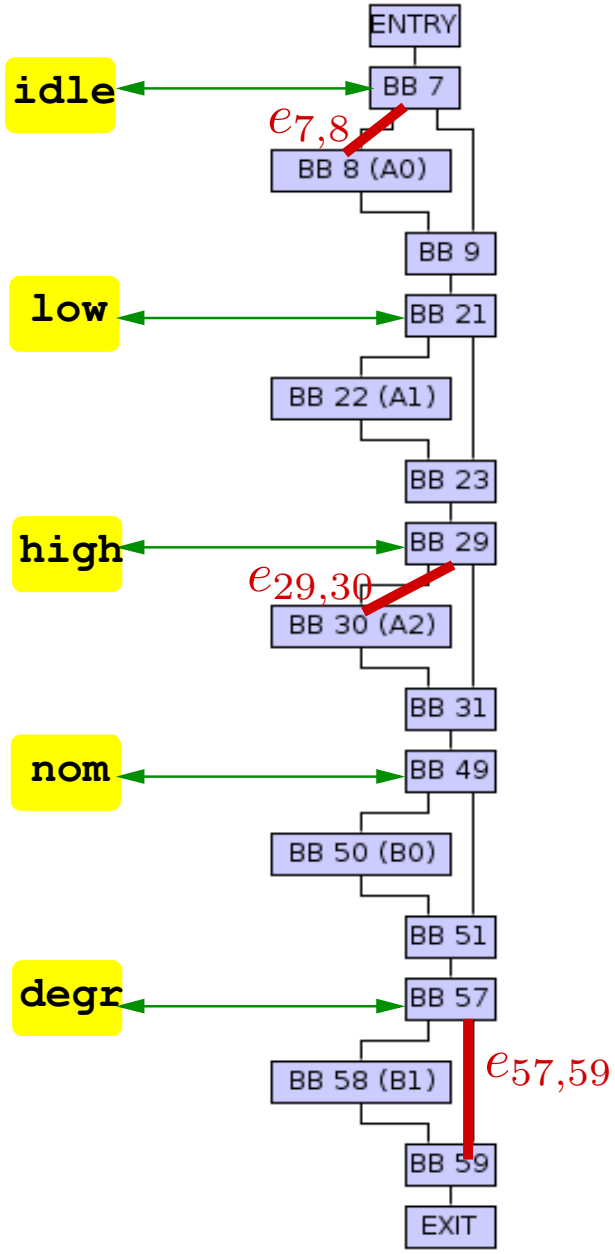
From (binary) paths to (HL) properties to (ILP) constraints

- Traceability has been achieved
- Feasibility of binary paths ?
  e.g. $e_{7,8}$ & $e_{29,30}$ & $e_{57,59}$



From (binary) paths to (HL) properties to (ILP) constraints

- Traceability has been achieved
- Feasibility of binary paths ?
  e.g. $e_{7,8}$ & $e_{29,30}$ & $e_{57,59}$
- Feasibitity as HL predicate:
  $\Phi = (\mathbf{idle} \wedge \mathbf{high} \wedge \neg \mathbf{degr})$

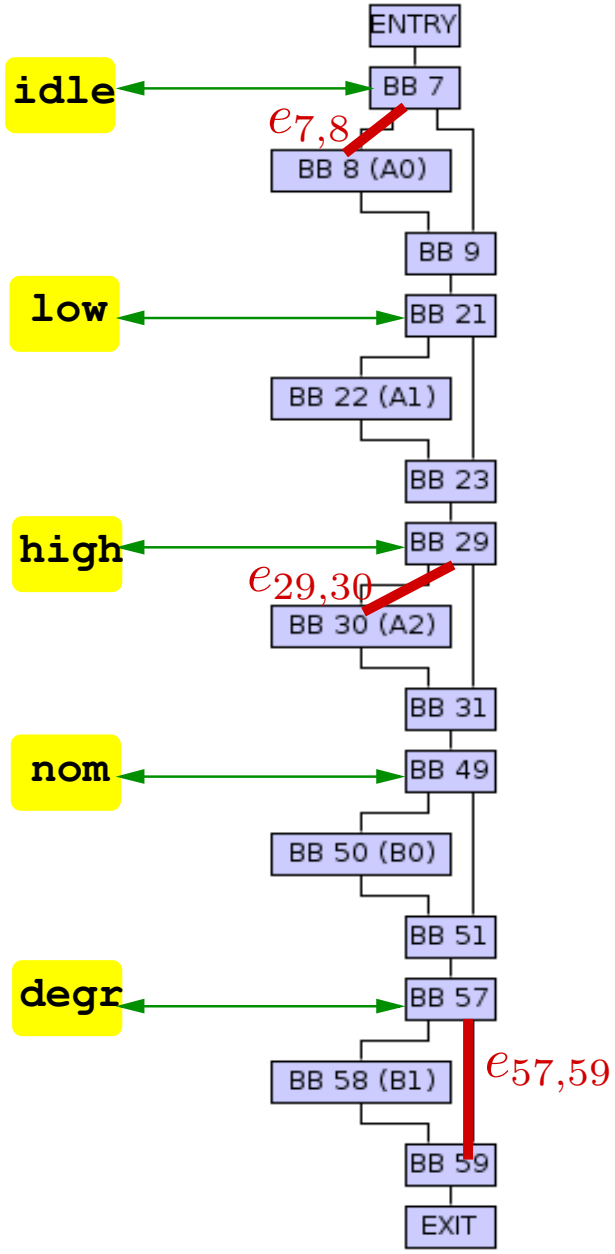


From (binary) paths to (HL) properties to (ILP) constraints

- Traceability has been achieved
- Feasibility of binary paths ?
  e.g. $e_{7,8}$ & $e_{29,30}$ & $e_{57,59}$
- Feasibitity as HL predicate:
  $\Phi = (\mathbf{idle} \wedge \mathbf{high} \wedge \neg \mathbf{degr})$
- Ask some HL verification tool:

  Is $\neg\Phi$ an invariant of the HL program ?

  (here: Lesar = Lustre model-checker)



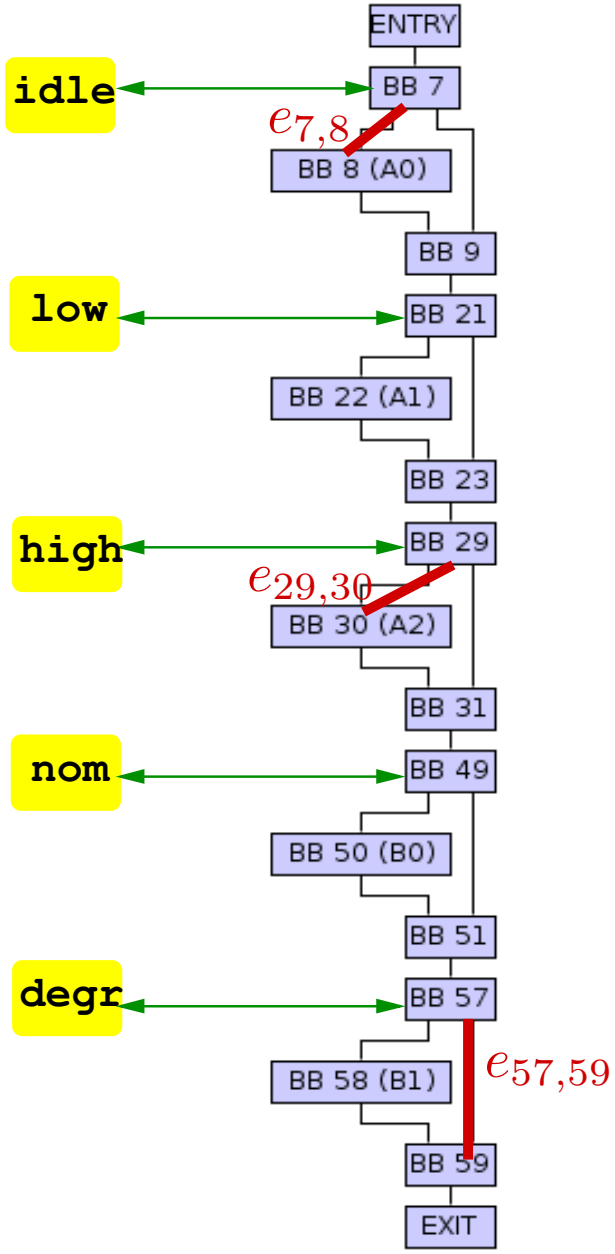
From (binary) paths to (HL) properties to (ILP) constraints

- Traceability has been achieved
- Feasibility of binary paths ?
  e.g. $e_{7,8}$ & $e_{29,30}$ & $e_{57,59}$
- Feasibitity as HL predicate:
  $\Phi = (\mathbf{idle} \wedge \mathbf{high} \wedge \neg \mathbf{degr})$
- Ask some HL verification tool:

  Is $\neg\Phi$ an invariant of the HL program ?

  (here: Lesar = Lustre model-checker)

  $\hookrightarrow$ Not proven, some path may be feasible...



From (binary) paths to (HL) properties to (ILP) constraints

- Traceability has been achieved
- Feasibility of binary paths ?
  e.g. $e_{7,8}$ & $e_{29,30}$ & $e_{57,59}$
- Feasibitity as HL predicate:
  $\Phi = (\mathbf{idle} \wedge \mathbf{high} \wedge \neg \mathbf{degr})$
- Ask some HL verification tool:

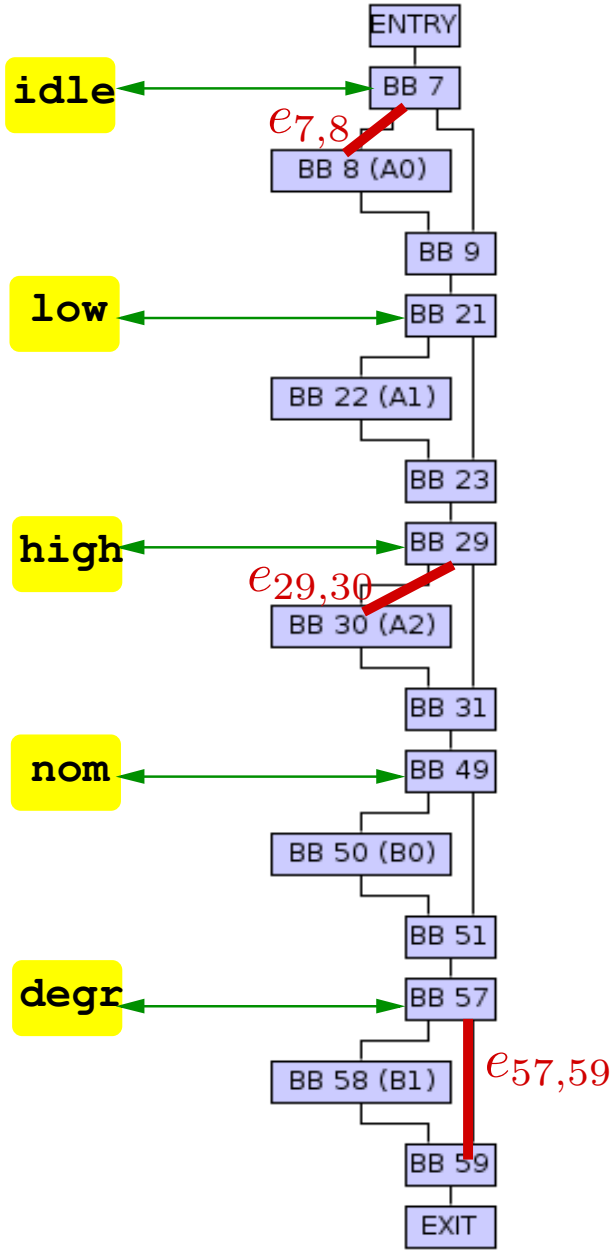
  Is $\neg\Phi$ an invariant of the HL program ?

  (here: Lesar = Lustre model-checker)

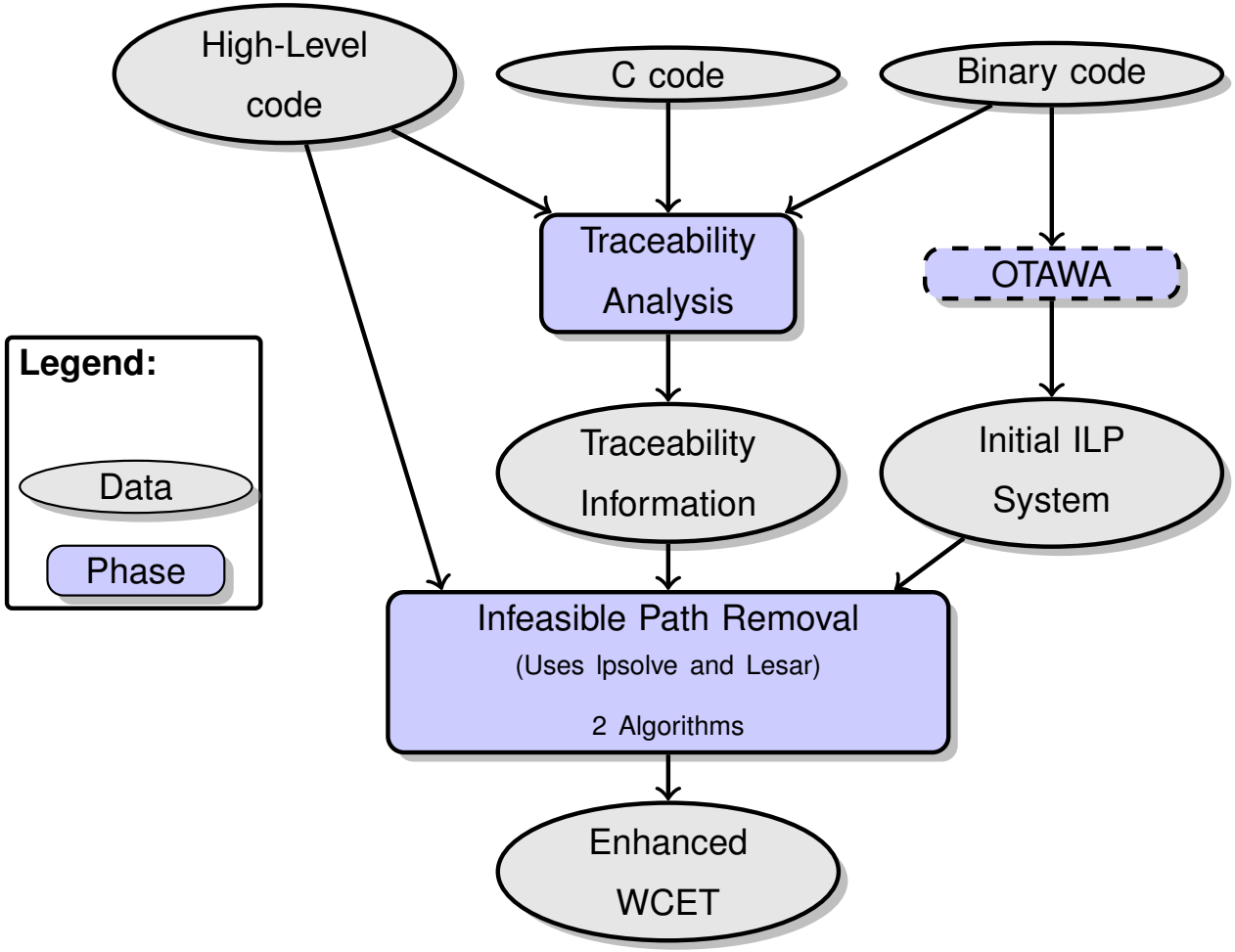
  $\hookrightarrow$ Not proven, some path may be feasible...

  $\hookrightarrow$ Proven. Infeasibility as ILP constraint:

  $e_{7,8} + e_{29,30} + e_{57,59} < 3$



Proof of concept workflow



- Core of the tool: Path Removal algorithm/strategy

Path removal

Iterative Method

- Algorithm:
  - ↪ (0) start with the initial (OTAWA) ILP problem, find a first worst-case path (WCP) candidate
  - ↪ (1) translate the WCP candidate into a HL predicate Φ
  - ↪ (2) ask the model-checker to prove that $\neg\Phi$ is an invariant:
    - ∗ Not Proven: the path is feasible a *best* estimation is reached
    - ∗ Proven: candidate WCP infeasible, add the corresponding constraint to the ILP, call LPSolve to find a new WCP candidate, return to step (2)

Path removal

Iterative Method

- Algorithm:
  - ↪ (0) start with the initial (OTAWA) ILP problem, find a first worst-case path (WCP) candidate
  - ↪ (1) translate the WCP candidate into a HL predicate Φ
  - ↪ (2) ask the model-checker to prove that $\neg\Phi$ is an invariant:
    - ∗ Not Proven: the path is feasible a *best* estimation is reached
    - ∗ Proven: candidate WCP infeasible, add the corresponding constraint to the ILP, call LPSolve to find a new WCP candidate, return to step (2)
- Pro: eventually results on the *optimal* WCET
- Cons: number of iterations combinatorial (paths removed one by one)

N.B. Optimality relative to model-checker capability

Path removal (cont'd)

Pairwise Stategies

- Empirical: most of interesting properties are pairwise disjunctions

  i.e. *implications*

  e.g. $\neg$**idle** $\vee$ $\neg$**low**, **degr** $\vee$ **idle**

- Idea: given a set of *interesting* HL conditions, check, a priori, all pairwise relations

  n.b. 4 relations for each pair $(x \vee y)$, $(x \vee \bar{y})$, $(\bar{x} \vee y)$, $(\bar{x} \vee \bar{y})$

- Add all the proven constraints to ILP system, call LPSolve once

- Pro: reasonable number of candidate properties (quadratic),

  leading to simple ILP constraints ($X + Y < 2$)

- Cons: *optimality* not guaranteed,

  some iterative steps necessary if optimality is targeted

Some results

- Iterative method

Bin Code		Wcet estimation		# iter.	cost (cpu seconds)			
opt.	# BB / edg.	initial	final		OTAWA	LPSolve	Lesar	Total
-O0	83 / 119	4726	2375	298	64s	90s	2.8s	163s
-O2	24 / 42	762	459	116	1s	1.5s	0.7s	4.5s

- Pairwise method

optim.	Wcet estimation		# pairs		cost (cpu seconds)			
	initial	final	checked	valid	OTAWA	LPSolve	Lesar	Total
-O0	4726	2376	144	21	64s	0.1s	1.4s	67
-O2	762	459	60	14	1s	0.1s	0.6s	2.3s

Conclusion

- A proof-of-concept tool / *representative* example

  ↪ n.b. however academic, Lustre is the core language of SCADE,
  actually used in industry (avionics, energy etc.)

- WCET enhancement *important* (as expected)

- Focus on *invariant* functional properties,
  hard/impossible to guess at low level

- More generally, *WCET* estimate may *benefit from* the specificities of Hard-RT
  design/compil methods