The W-SEPT project[1]: Towards Semantic-aware WCET Estimation

C. Maiza[2], P. Raymond[2], C. Parent-Vigouroux[2], A. Bonenfant[3], F. Carrier[2], H. Cassé[3], P. Cuenot[5], D. Claraz[5], N. Halbwachs[2], E. Jahier[2], H .Li[4], Mi. De Michiel[3], V. Mussot[3], I. Puaut[4], C. Rochange[3], E. Rohou[4], J. Ruiz[3], P. Sotin[3], W-T. Sun[3]

WCET Workshop, 2017

[1] This project was founded by ANR

[2] Univ. Grenoble Alpes-VERIMAG

[3] Univ. Toulouse-IRIT

[4] Univ. Rennes 1/INRIA-IRISA

[5] Continental

Introduction

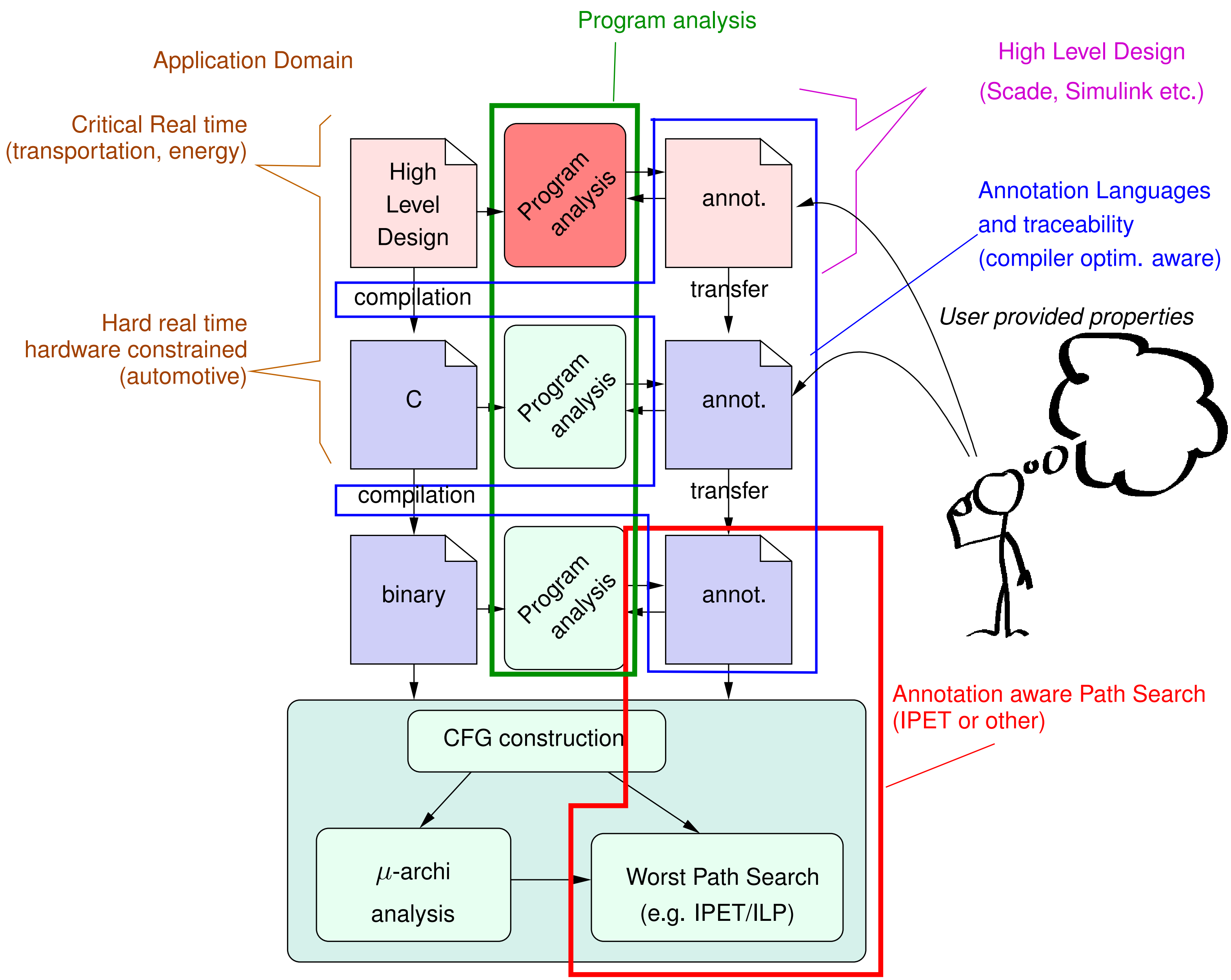
General goal

- Increase the precision of WCET estimation...
  - $\hookrightarrow$ by focusing on the influence of the software semantics
  - $\hookrightarrow$ i.e., not about hardware modeling !

The project

- W-SEPT $=$ WCET: SEmantics, Precision, Traceability
- Find, trace and exploit semantics information:
  - $\hookrightarrow$ Main issue: Express infeasible paths given or automatically discovered and preserve them through compilation process
  - $\hookrightarrow$ Approach:
    - $*$ Use a common format: FFX (Flow fact Format Xml-like)
    - $*$ Rely/adapt/extend existing timing analysis tool (OTAWA)

Project overview



Partners: skills/complementarity

- IRIT Toulouse: C/binary level and WCET computation
- Verimag Grenoble: high-level/design, semantic analysis
- Inria/Irisa Rennes: compilation and binary level
- Continental Toulouse: industrial application - Engine Management System (EMS)

This talk

- Focus on 3 topics/experiments
  - ↪ Exploiting High Level Properties
  - ↪ Tracing flow information through compiler optimization
  - ↪ Expressing and exploiting path properties

Exploiting High Level Properties

Model-based design

- Particularly in safety critical domains (Scade/Lustre)
- But not only (Simulink/Stateflow)
- Compilation process: HL to C to bin

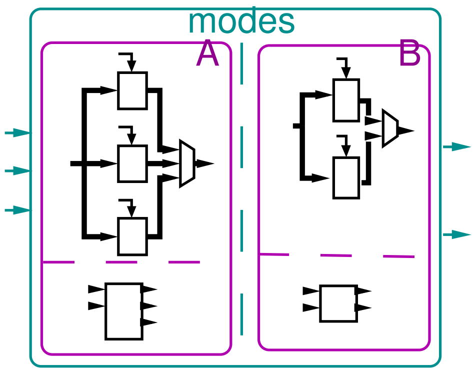
Consequences for timing analysis

- Semantic static analysis exist at HL
- HL properties may have strong influence on WCET
- HL properties are “hard” to discover at lower level

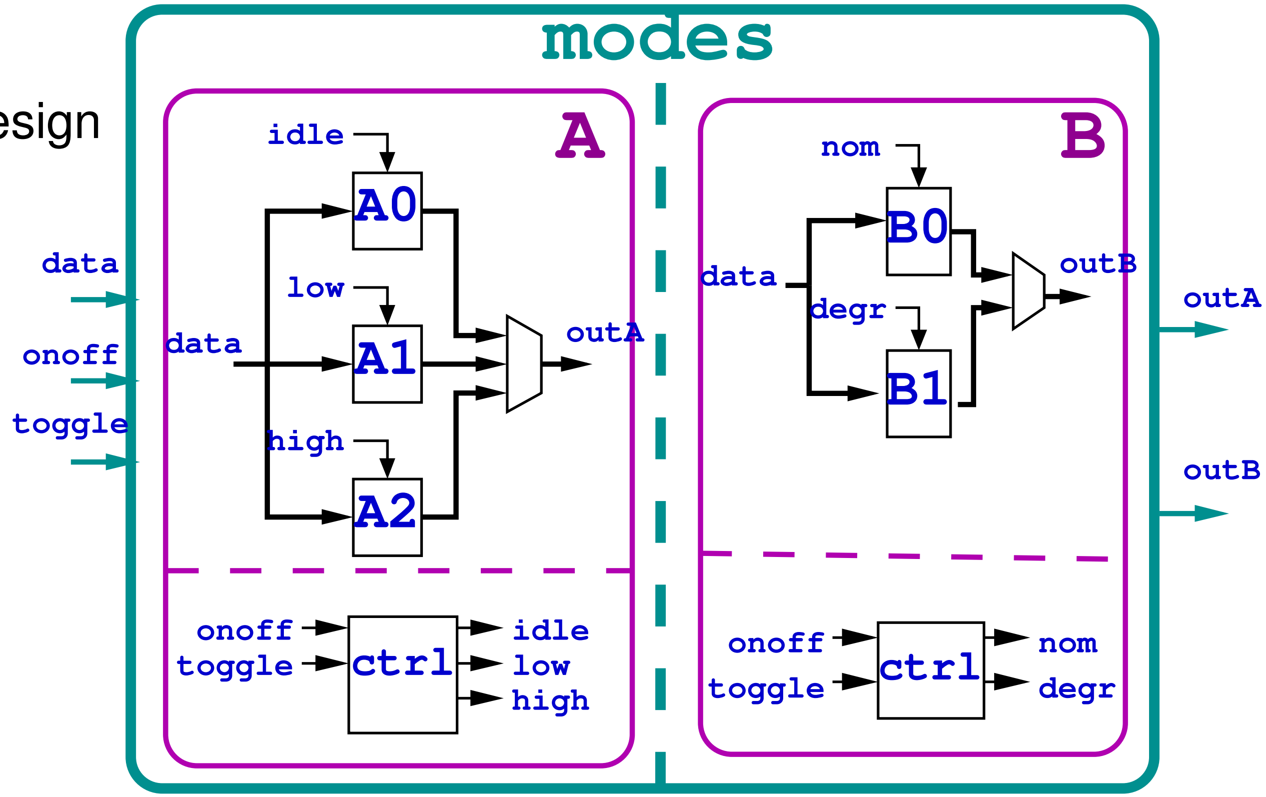
Experiment with Lustre

- Representative: Lustre $\sim$ Scade (avionics)
- ... and not so different from Simulink
- What is important: *synchronous paradigm*, i.e., execution $=$ infinite loop, each iteration performs an atomic reaction

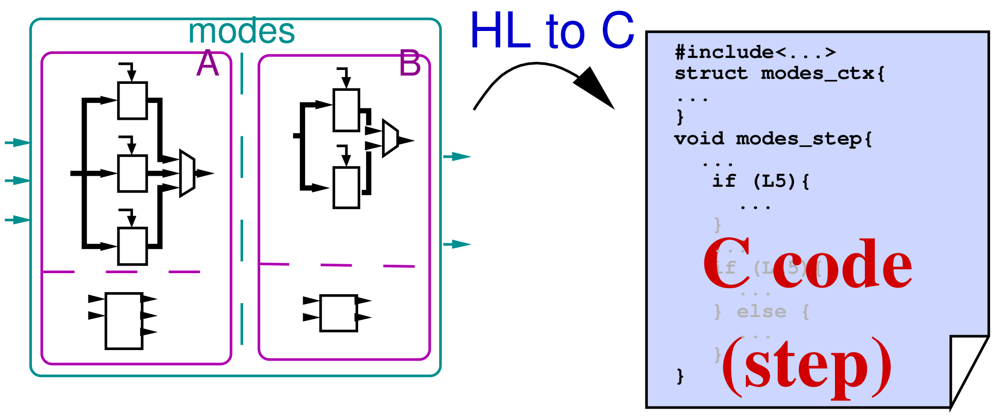
Synchronous Programming Workflow



- Design level:
  - ↪ Concurrent, Hierarchic design
  - ↪ Idealized Concurrency
  - ↪ Behavior = sequence of reactions logical discrete time
  - ↪ Several styles/languages Here: data-flow/Lustre



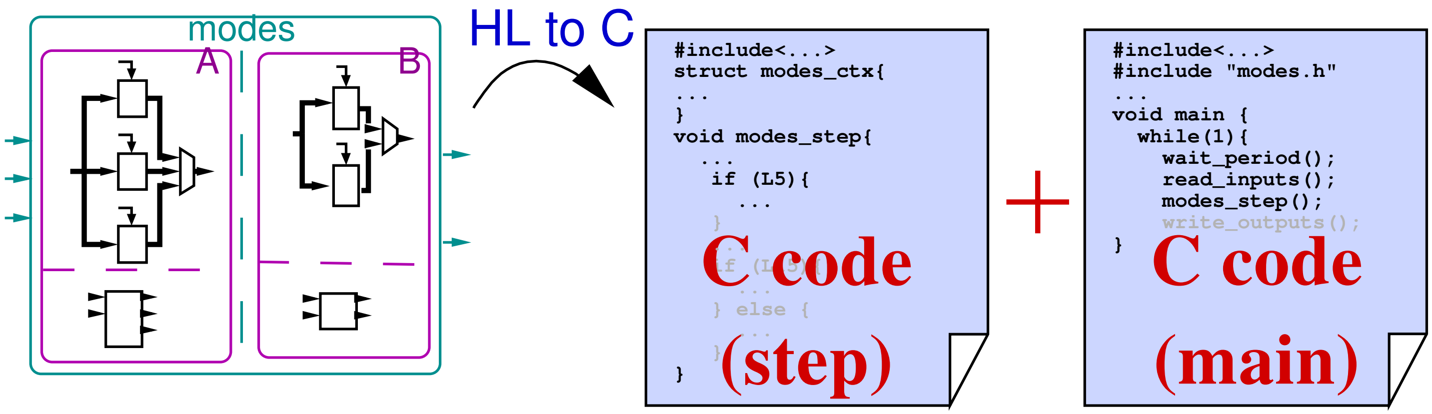
Synchronous Programming Workflow



- Synchronous Compiler
  - ↪ Target language = C
  - ↪ Generates the step procedure (+ the necessary memory/ctx)
  - ↪ Basically: no more concurrency (static scheduling)
  - ↪ Simple sequential code

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

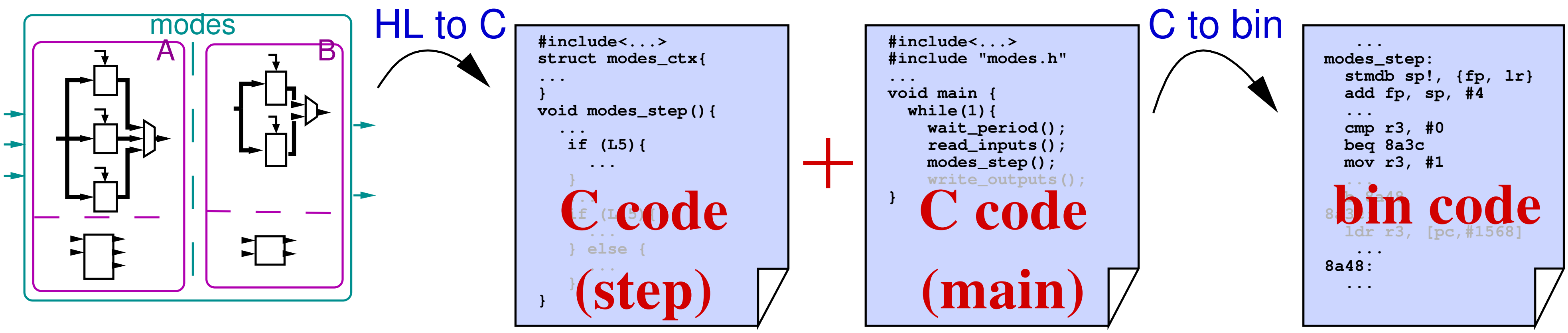
Synchronous Programming Workflow



- 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

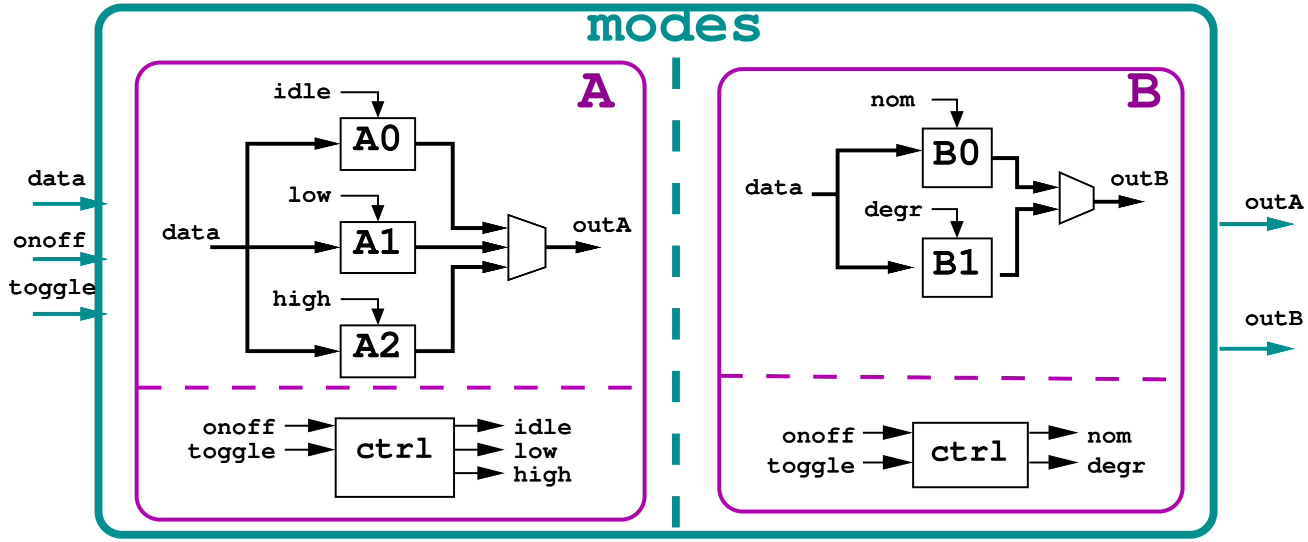


- Binary code
  - ↪ via arm-elf-gcc
  - ↪ WCET estimation should be done here

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

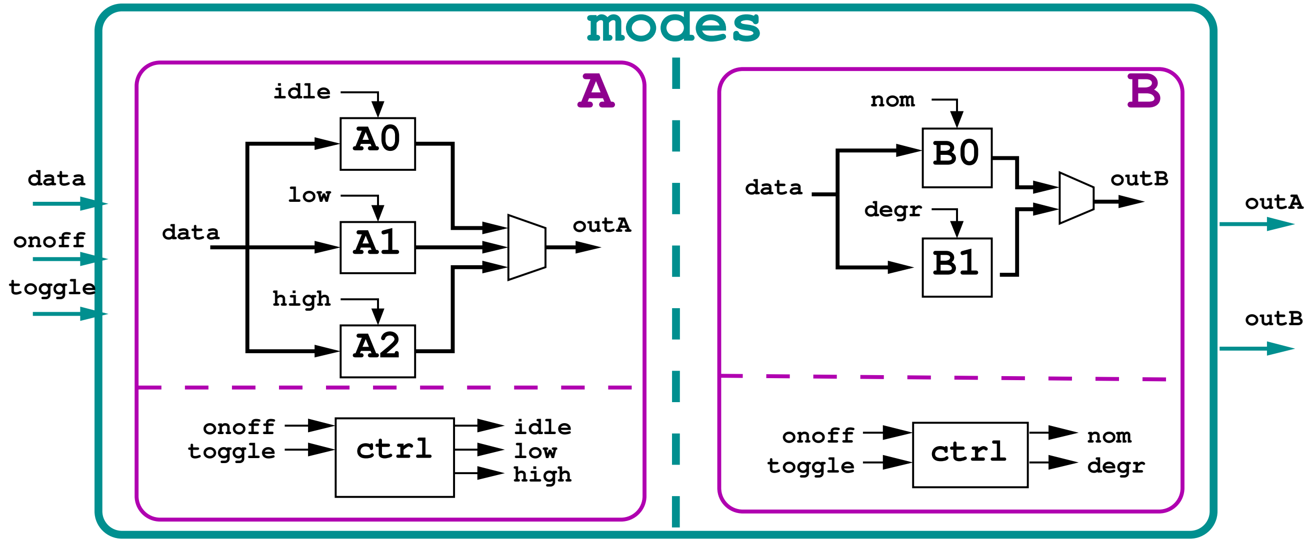
```
  ...
modes_step:
  stmdb sp!, {fp, lr}
  add fp, sp, #4
  ...
  cmp r3, #0
  beq 8a3c
  mov r3, #1
  ...
  b 8a48
8a3c:
  ldr r3, [pc,#1568]
  ...
8a48:
  ...
```

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 the C or bin code of one step

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`**

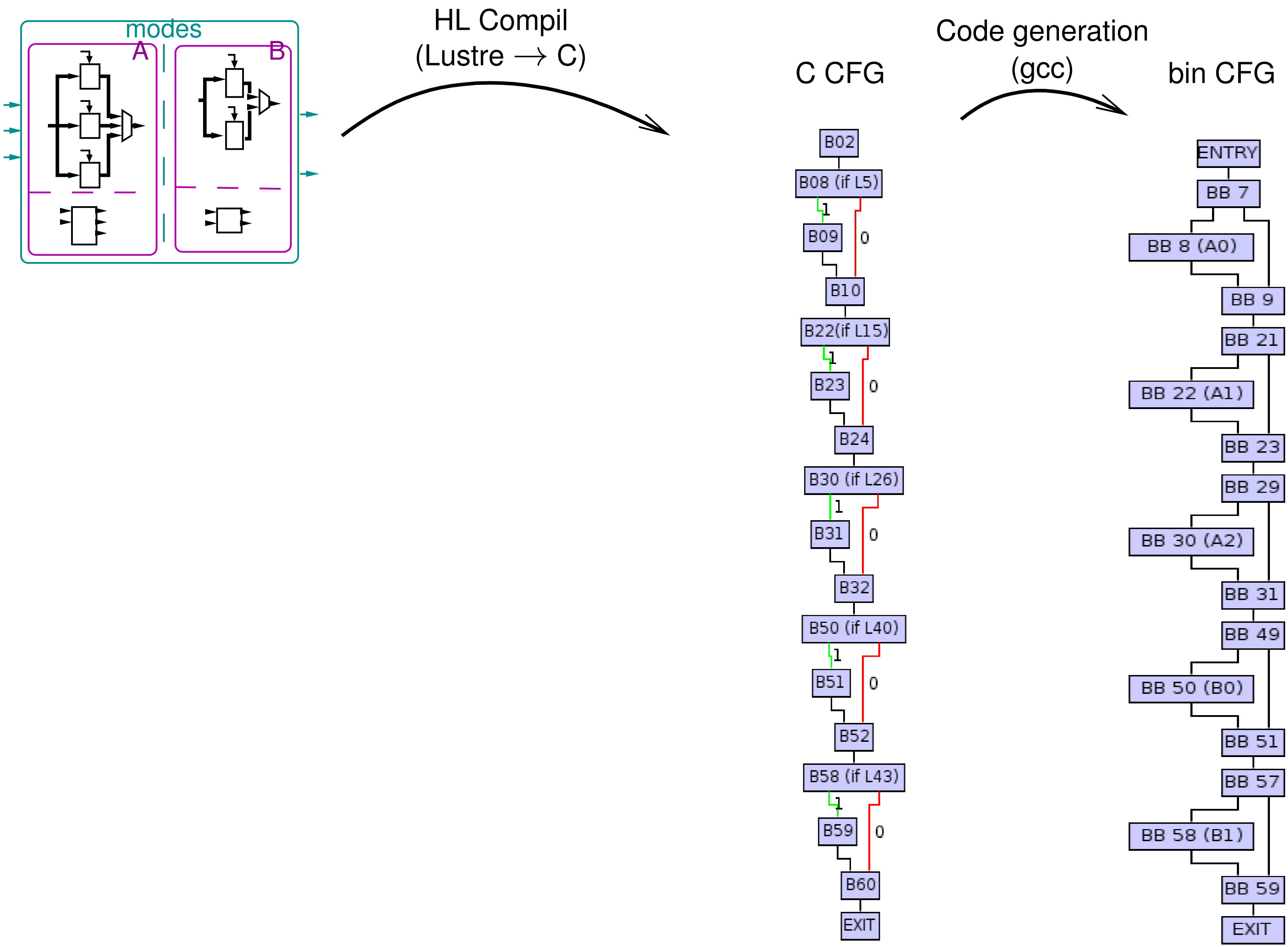
  ↪ no chance to be obvious on the code
- In all cases, relatively complex properties:

  ↪ infinite loop invariants

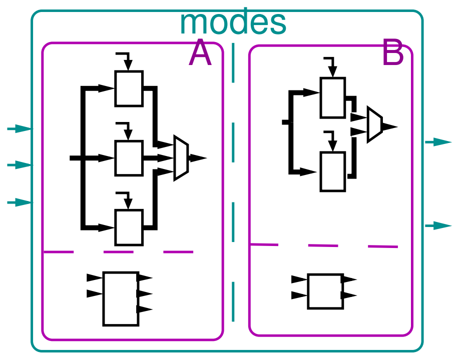
  ↪ unlikely to be discovered by analysing the C or bin code of one step
- Can be discovered using, e.g., model-checking techniques

  (here, Lesar = Lustre Model-Checker)

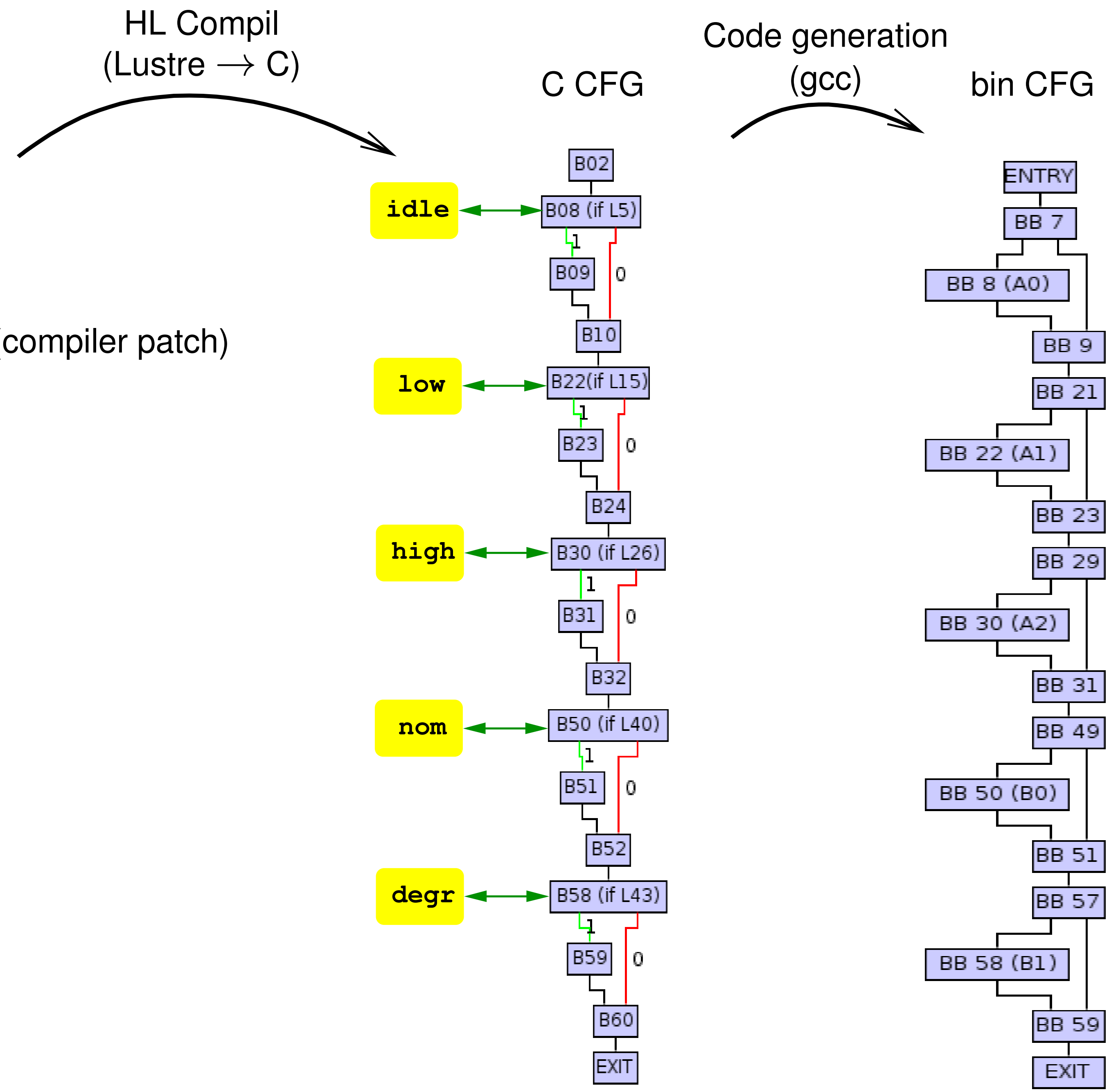
Traceability: form HL property to binary (ILP) constraint



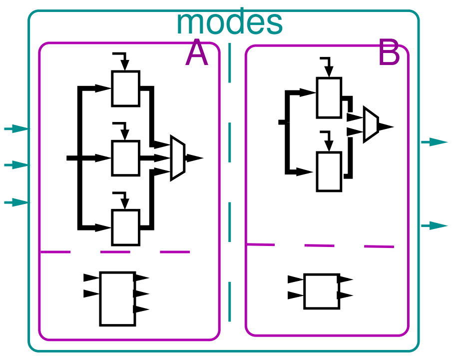
Traceability: form HL property to binary (ILP) constraint



- Relate HL var to C var (compiler patch)

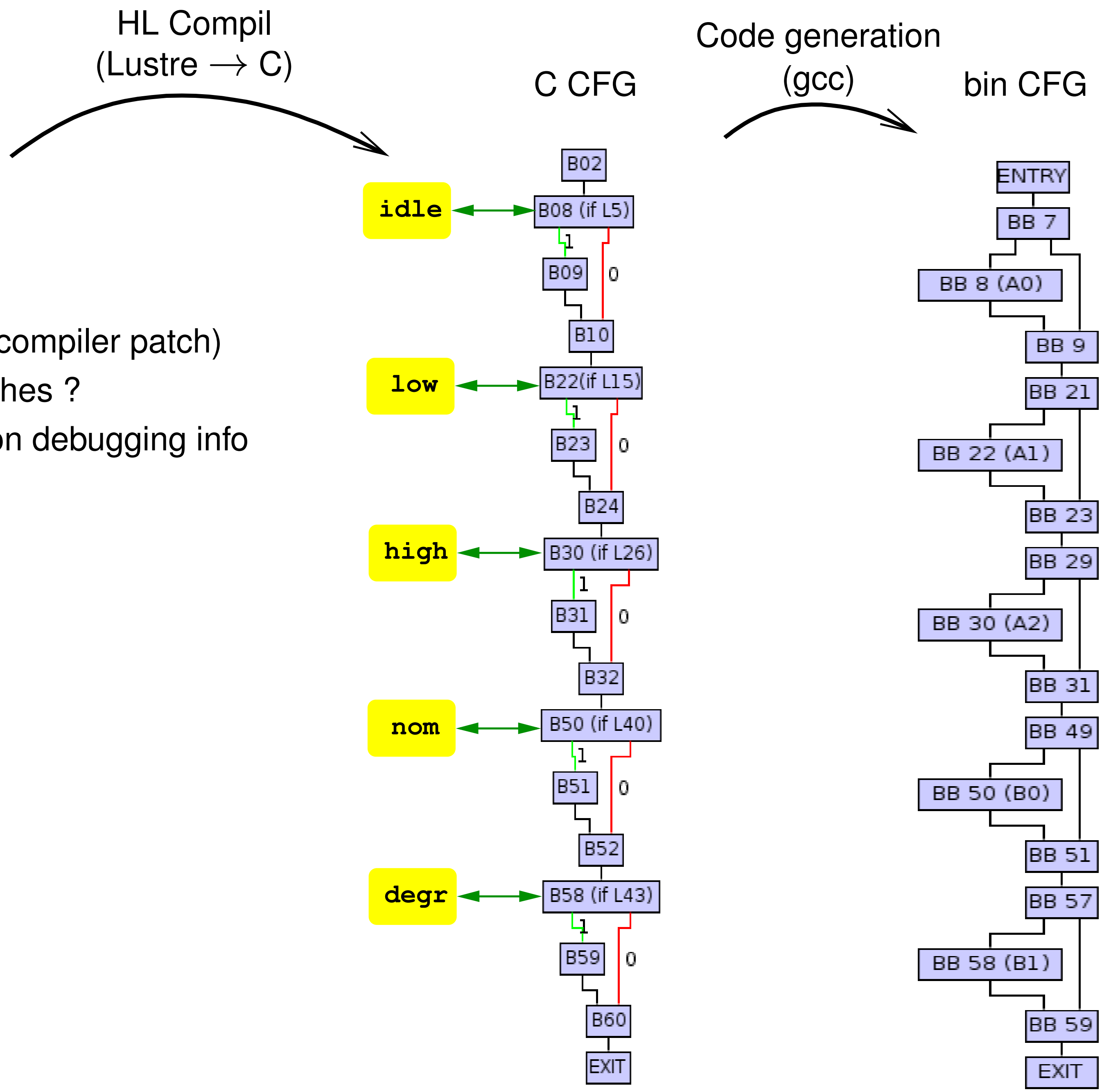


Traceability: form HL property to binary (ILP) constraint

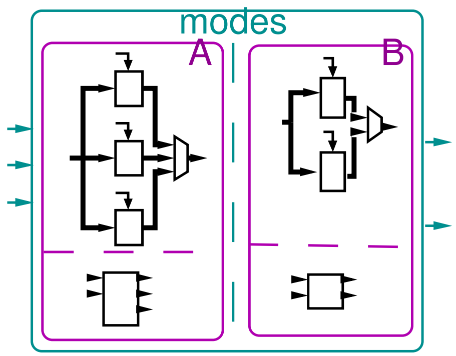


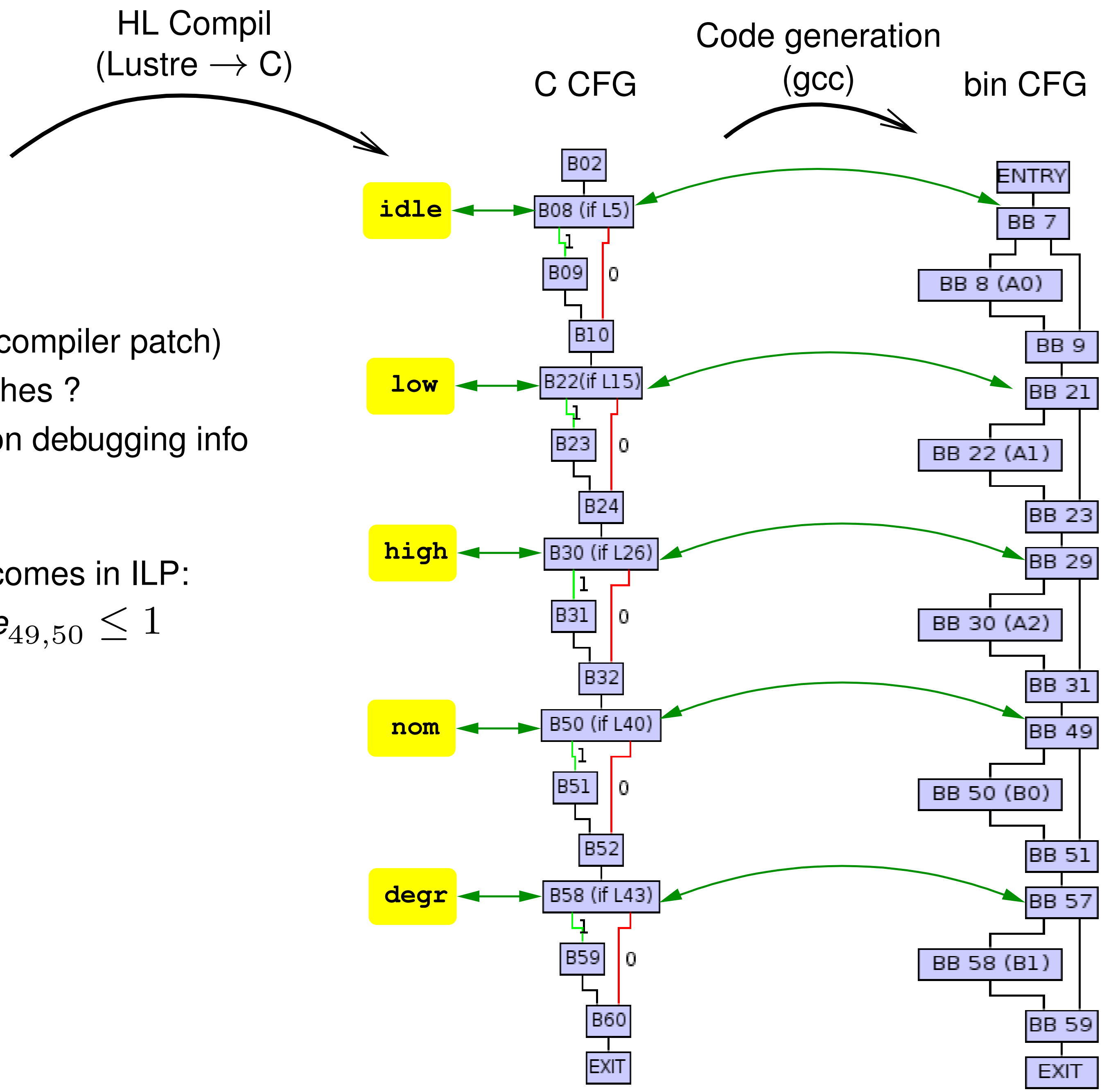
- Relate HL var to C var (compiler patch)
- C branches to bin branches ?

↪ simple heuristic: rely on debugging info



Traceability: form HL property to binary (ILP) constraint



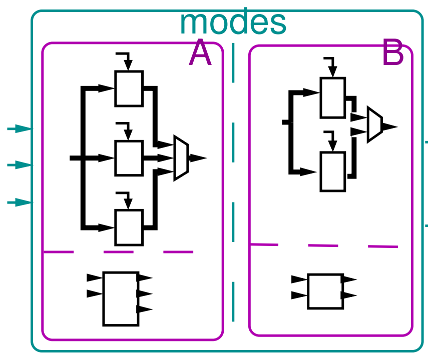


- Relate HL var to C var (compiler patch)
- C branches to bin branches ?

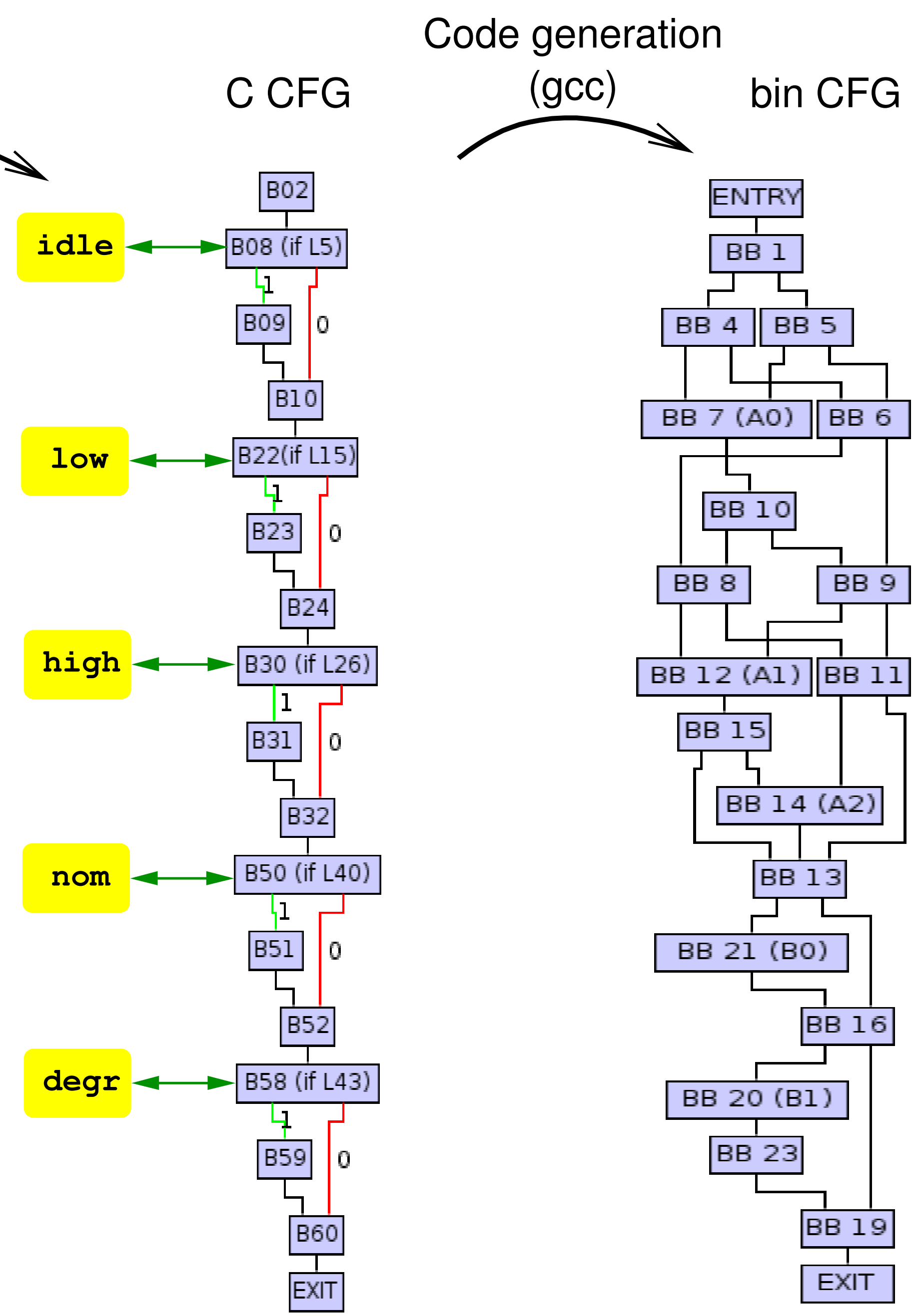
$\hookrightarrow$ simple heuristic: rely on debugging info

- No optimization ($-$O0)
  CFG's strictly match
  e.g. $\neg(\texttt{high} \wedge \texttt{nom})$ becomes in ILP:
  $$\textit{edge}_{29,30} + \textit{edge}_{49,50} \leq 1$$

Traceability: form HL property to binary (ILP) constraint



HL Compil
(Lustre $\rightarrow$ C)

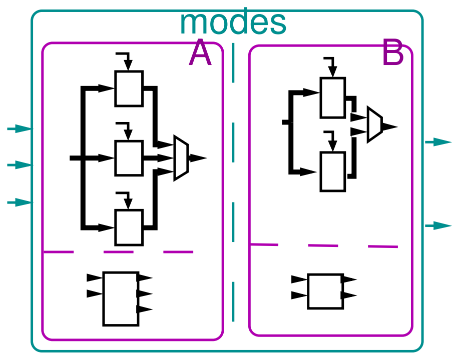


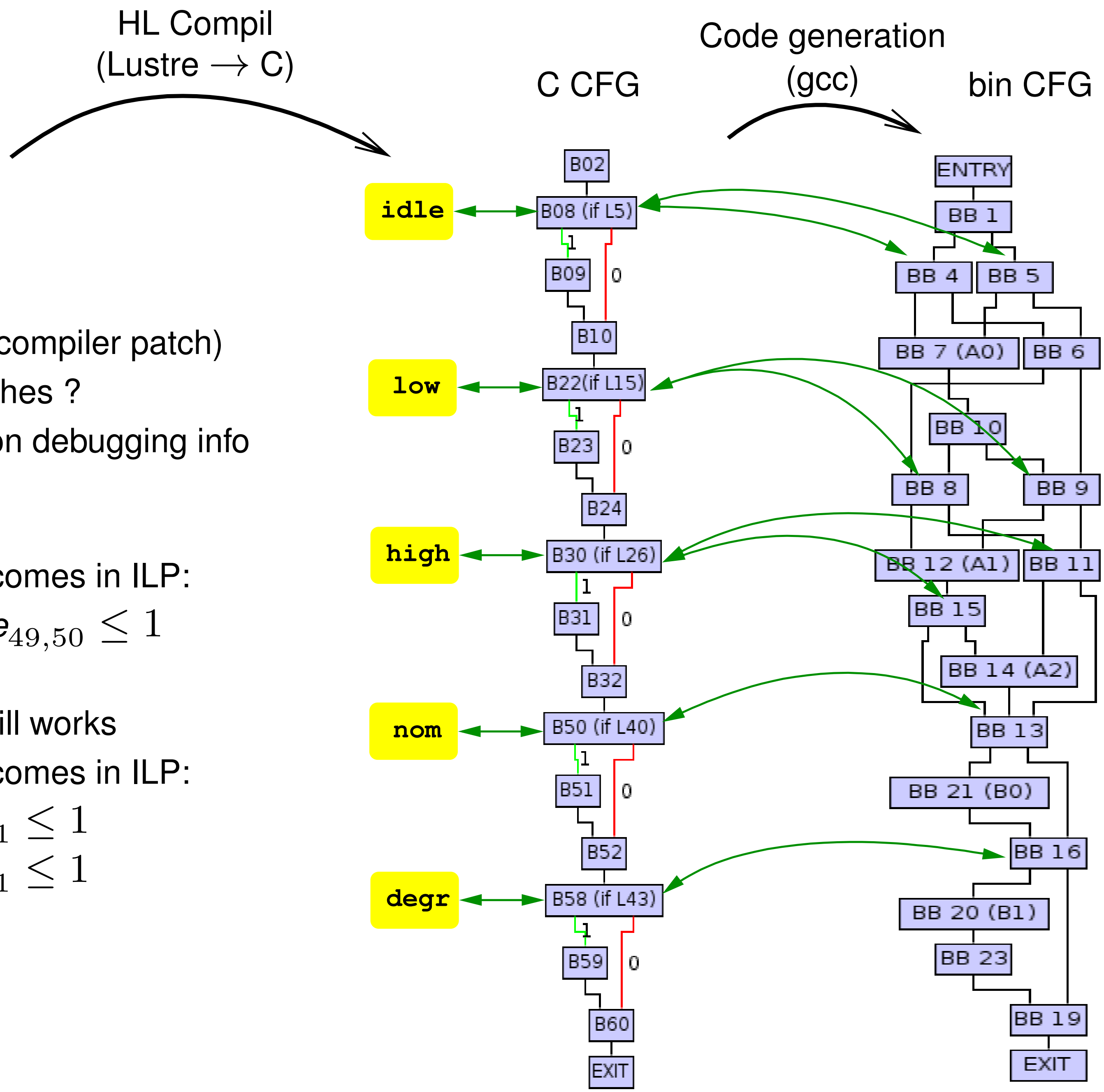
- Relate HL var to C var (compiler patch)
- C branches to bin branches ?

$\hookrightarrow$ simple heuristic: rely on debugging info

- No optimization ($-\mathtt{O0}$)
  CFG's strictly match
  e.g. $\neg(\mathtt{high} \wedge \mathtt{nom})$ becomes in ILP:
  $$edge_{29,30} + edge_{49,50} \leq 1$$
- Optimization ($-\mathtt{O2}$)
  CFG obfuscated, but still works

Traceability: form HL property to binary (ILP) constraint





- Relate HL var to C var (compiler patch)
- C branches to bin branches ?

$\hookrightarrow$ simple heuristic: rely on debugging info

- No optimization ($-\mathtt{O0}$)

  CFG's strictly match

  e.g. $\neg(\texttt{high} \land \texttt{nom})$ becomes in ILP:

  $$\mathit{edge}_{29,30} + \mathit{edge}_{49,50} \leq 1$$

- Optimization ($-\mathtt{O2}$)

  CFG obfuscated, but still works

  e.g. $\neg(\texttt{high} \land \texttt{nom})$ becomes in ILP:

  $$\mathit{edge}_{15,14} + \mathit{edge}_{13,21} \leq 1$$
  $$\mathit{edge}_{11,14} + \mathit{edge}_{13,21} \leq 1$$

High Level properties, conclusion

- Fully automatic proof of concept
- Implements 2 strategies:
  - ↪ Iterative: computes a WCET candidate, try to refute it with HL model checking, and so on until WCET candidate cannot be refuted.
    - ∗ reaches a (relative) best solution, but converges very slowly
  - ↪ Pairwise a priori: check, once for all, any possible pairwise relation between “well chosen” HL variables
    - ∗ e.g. clocks are clearly good candidates
    - ∗ quadratic number of relations to check, but single WCET analysis
- Experiments: with both strategy, the gain is about 40%, pairwise strategy runs much faster (few seconds vs few minutes).

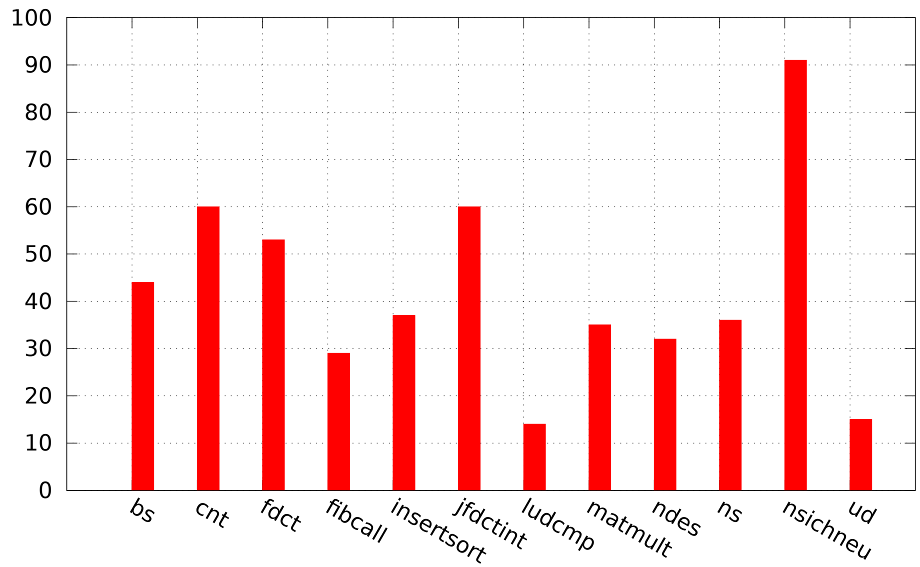
⋆ P. Raymond, C. Maiza, C. Parent-Vigouroux, F. Carrier, and . Asavoae.
Timing analysis enhancement for synchronous program. *Real-Time Systems*, 2015.

Problem

- Infeasible path properties are generally discovered/given at C level
- Relate infeasible *C path* to infeasible *binary path* ?
- Radical solution: No optimization: perfect match, no problem...

  But the code is likely to be rather inefficient!
- Impact of optimization on WCET estimation, for 12 classical benchmarks:

  ↪ "`-O1` code" WCET as a % of "`-O0` code" WCET



Allow optimization in WCET estimation?

- Rely on existing compiler tracing facilities (e.g. dwarf)
  - ↪ Accept to lose some properties (cf. previous topic)
- Allow optimization that do not (or slightly) impact the CFG
  - ↪ not so bad: data optim. largely speedup code in general
- Modify/adapt compilers to make them trace-property aware.
  - ↪ Probably the most satisfactory ...
  - ↪ .. but requires a lot of work
  - ↪ Not suitable when off-the-shelf, black-box compilers are required

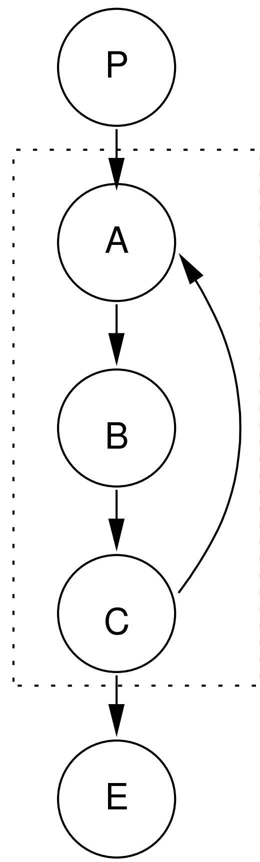
The project approach

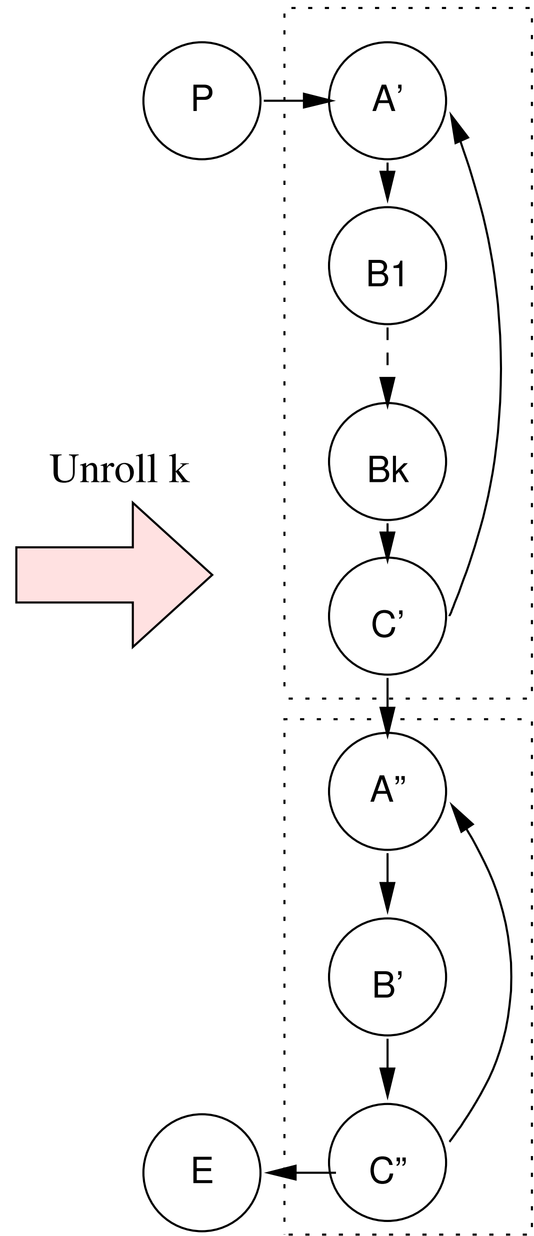
- Study the “path-aware” compiler approach
- Experiment/proof-of-concept based on the LLVM compilation platform

General idea

- Flow informations = IPET-like constraints
- CFG transformation = constraint rewriting
  - $\hookrightarrow$ possible loss in precision
- Example: loop bounds and loop unrolling
  - $\hookrightarrow$ #A $\leq$ Xmax
  - $\hookrightarrow$ Becomes:
    - $*$ #A' $\leq$ Xmax / k
    - $*$ and #A" $\leq$ k -1
- Proof of concept for $\sim$ 10 classical optim.
- Results for a Lustre program, with and without infeasible path search and tracing:

Analysis & tracing	optim. level `-O0`	`-O1`	`-O2`
Off	2896 (100%)	1523 (52.5%)	1542 (53.2%)
On	2014 (69.5%)	997 (34.4%)	998 (34.5%)





$\star$ Li H., Puaut I. and Rohou E.
Traceability of Flow Information: Reconciling Compiler Optimizations and WCET Estimation. *RTNS'14*

Introduction

- How to tell to the WCET analyser that some paths are infeasible ?
- Basically two kinds of methods:

$\hookrightarrow$ Make infeasibility explicit, via CFG transformation:

  - $\ast$ can (virtually) handle any property ...
  - $\ast$ ... but beware of graph size explosion !

$\hookrightarrow$ keep infeasibility implicit, via additional IPET constraint

  - $\ast$ “ideally” compact (in fact, complexity is transfered to ILP solver)
  - $\ast$ ... but possible loss in precision

Introduction

- How to tell to the WCET analyser that some paths are infeasible ?
- Basically two kinds of methods:

↪ Make infeasibility explicit, via CFG transformation:

  - can (virtually) handle any property ...
  - ... but beware of graph size explosion !

↪ keep infeasibility implicit, via additional IPET constraint

  - “ideally” compact (in fact, complexity is transfered to ILP solver)
  - ... but possible loss in precision

- Or maybe a mix of both ?

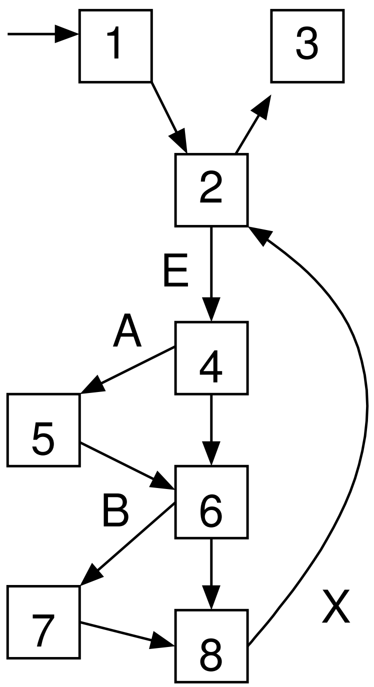
The project approach

- Design a versatile formalism, mixing explicit and implicit features
- PPA (Path Property Automata):

  $\hookrightarrow$ Inherits from formal language theory:

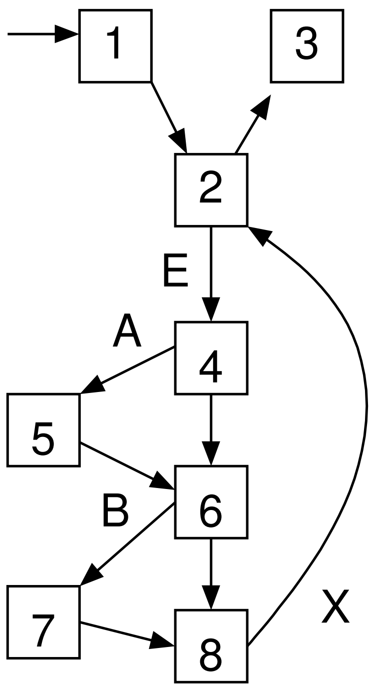
  - a CFG (program) $\Leftrightarrow$ a language whose words are the executions
  - a property $\Leftrightarrow$ an automaton recognizing feasible paths
  - removing infeasible path $\Leftrightarrow$ intersecting the CFG and the property
  - use hierarchic automata (rather than flat ones) for concision

Example (explicit product)



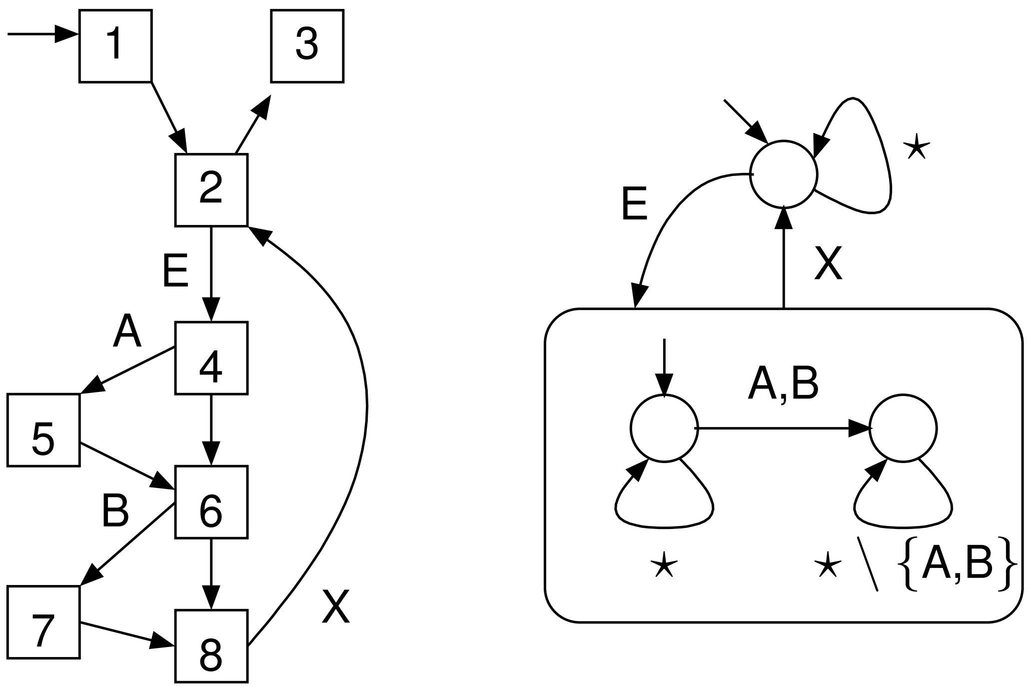
- Program CFG, an execution $=$ a word over alphabet $\{$E,A,B,X$\}$

Example (explicit product)



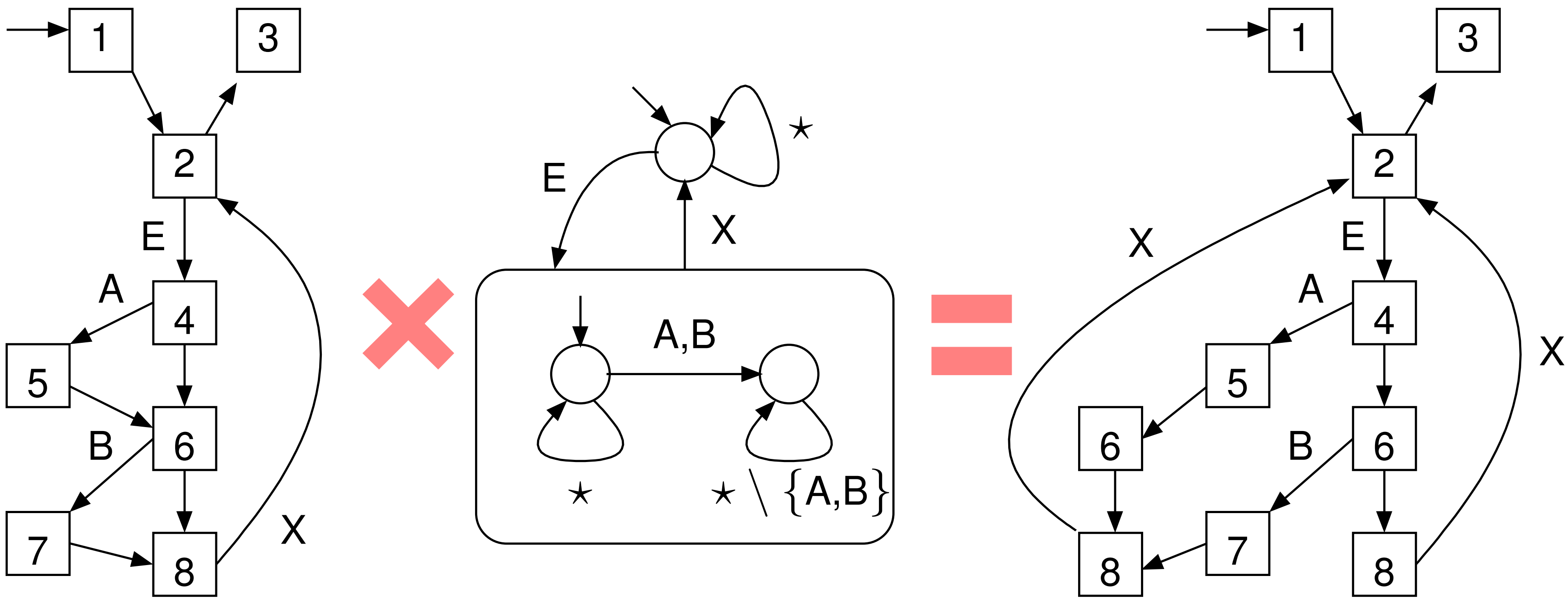
- Program CFG, an execution = a word over alphabet {E,A,B,X}
- Informal property: A and B exclusive at each iteration

Example (explicit product)



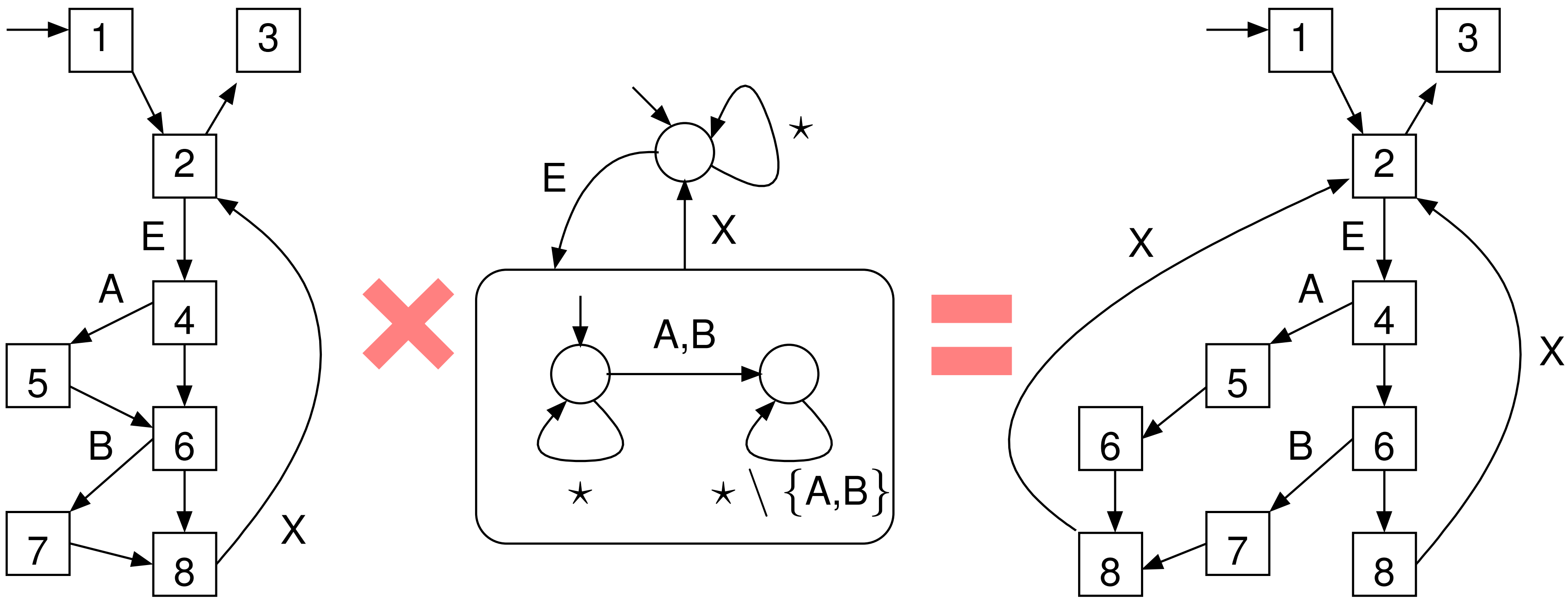
- Program CFG, an execution $=$ a word over alphabet $\{$E,A,B,X$\}$
- Informal property: A and B exclusive at each iteration
- Formalized as a language recognizer (PPA syntax)

Example (explicit product)



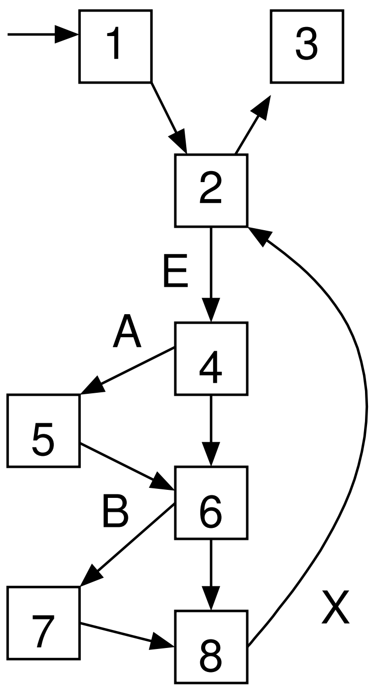
- Program CFG, an execution $=$ a word over alphabet $\{$E,A,B,X$\}$
- Informal property: A and B exclusive at each iteration
- Formalized as a language recognizer (PPA syntax)
- CFG $\times$ PPA product $\Leftrightarrow$ language intersection

Example (explicit product)



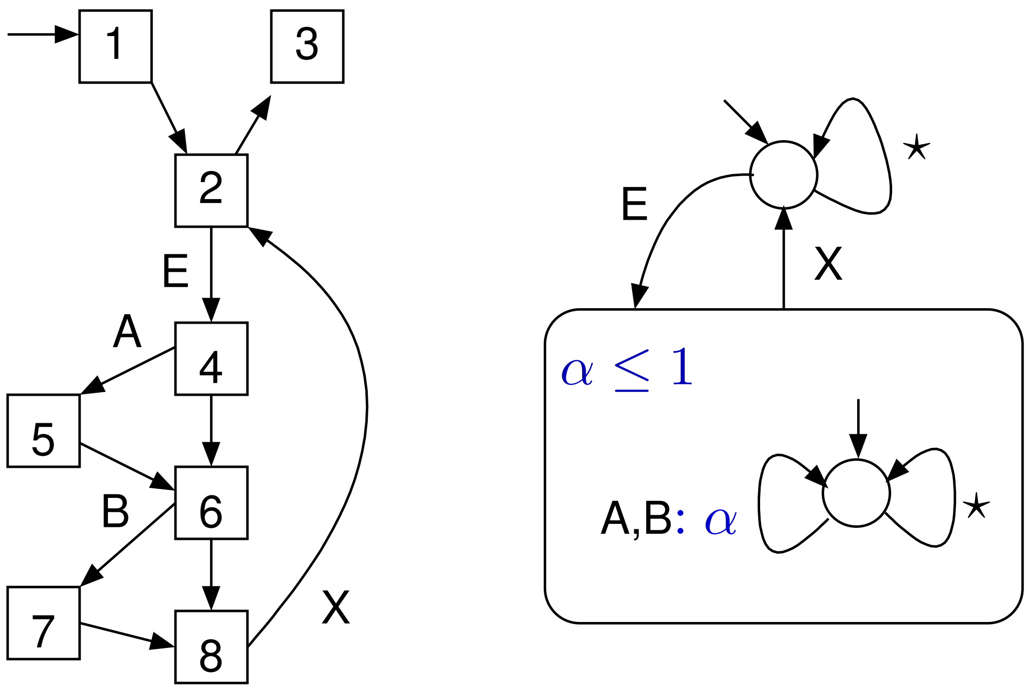
- Program CFG, an execution $=$ a word over alphabet $\{$E,A,B,X$\}$
- Informal property: A and B exclusive at each iteration
- Formalized as a language recognizer (PPA syntax)
- CFG $\times$ PPA product $\Leftrightarrow$ language intersection
- Explicit approach: beware of graph size explosion !

Example (mixed explicit/implicit product)



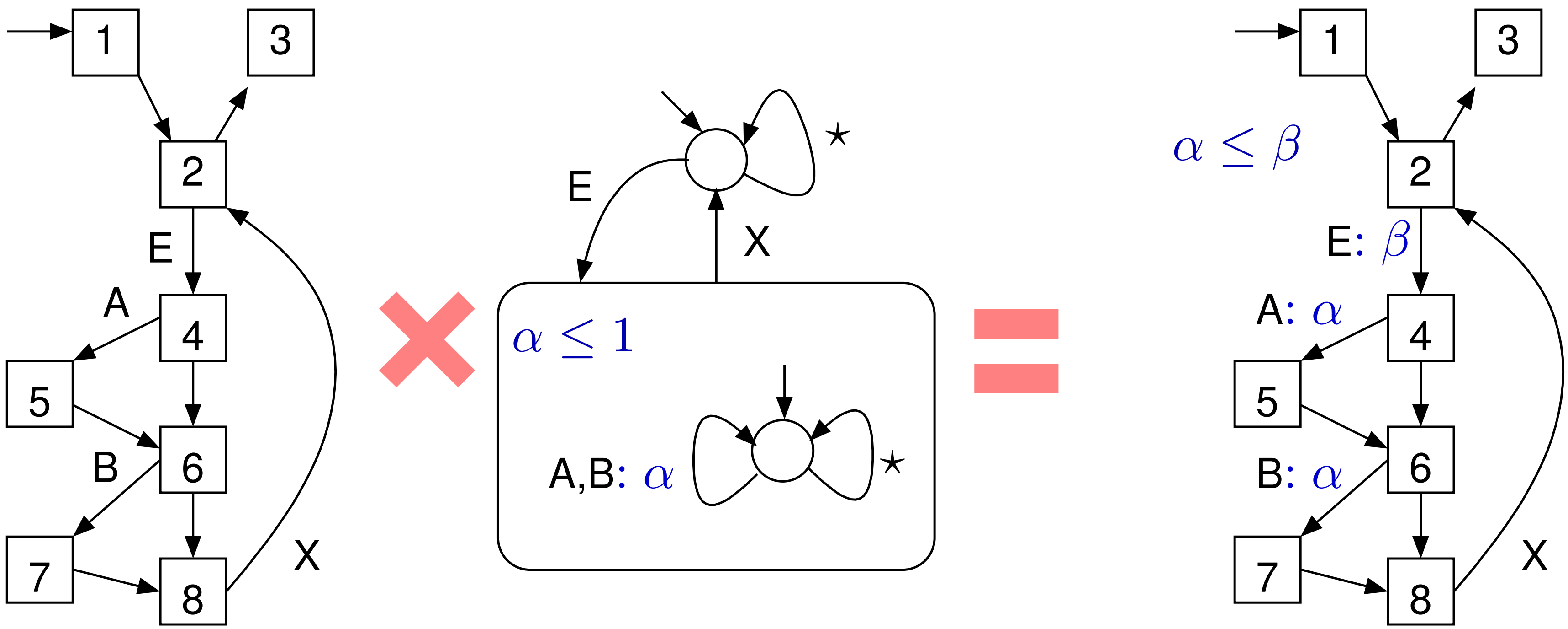
- Program CFG, an execution = a word over alphabet $\{$E,A,B,X$\}$
- Informal property: A and B exclusive at each iteration

Example (mixed explicit/implicit product)



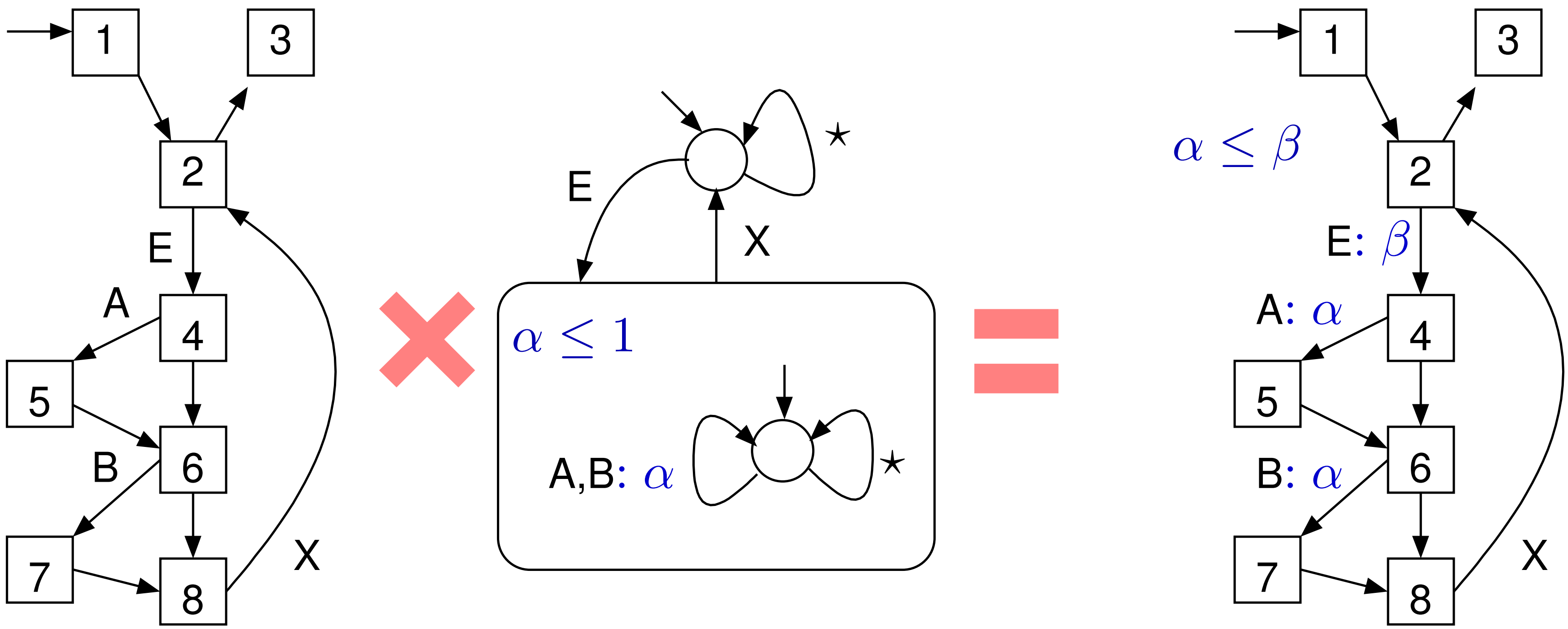
- Program CFG, an execution $=$ a word over alphabet $\{$E,A,B,X$\}$
- Informal property: A and B exclusive at each iteration
- Language recognizer **with local counters and constraints** (PPA syntax)

Example (mixed explicit/implicit product)



- Program CFG, an execution $=$ a word over alphabet $\{$E,A,B,X$\}$
- Informal property: A and B exclusive at each iteration
- Language recognizer **with local counters and constraints** (PPA syntax)
- Extended counter-aware product $\Leftrightarrow$ CFG + ILP constraints

Example (mixed explicit/implicit product)



- Program CFG, an execution = a word over alphabet {E,A,B,X}
- Informal property: A and B exclusive at each iteration
- Language recognizer **with local counters and constraints** (PPA syntax)
- Extended counter-aware product ⇔ CFG + ILP constraints
- Mixed explicit/implicit approach

⋆ Mussot V. and Sotin P.
Improving WCET Analysis Precision through Automata Product. *RTCSA*, 2015.

Conclusion

- Other topics studied/started during the project:
  - ↪ Semantic analysis at binary level
  - ↪ Limits of IPET/ILP methods
  - ↪ Beyond ILP: semantic + timing analysis as a whole
  - ↪ Targeting “costly” part of program (branch deltas)
  - ↪ User-guided analysis
  - ↪ etc. see `http://wsept.inria.fr`
- General result: a semantic-awre WCET workflow
- Raised interest from industrial partern

Conclusion

- Other topics studied/started during the project:
  - ↪ Semantic analysis at binary level
  - ↪ Limits of IPET/ILP methods
  - ↪ Beyond ILP: semantic + timing analysis as a whole
  - ↪ Targeting “costly” part of program (branch deltas)
  - ↪ User-guided analysis
  - ↪ etc. see `http://wsept.inria.fr`
- General result: a semantic-awre WCET workflow
- Raised interest from industrial partern

Thanks for your attention !

Questions ?