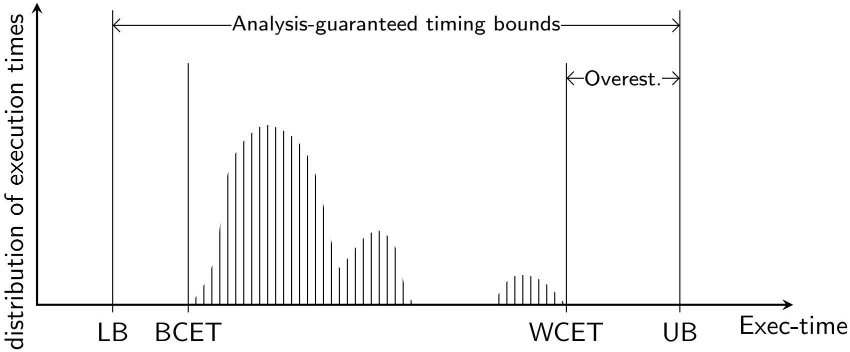
W-SEPT ANR project (W-7): WCET and synchronous program

Claire Maiza, Pascal Raymond

Synchron 2012

W-7: back to semantics

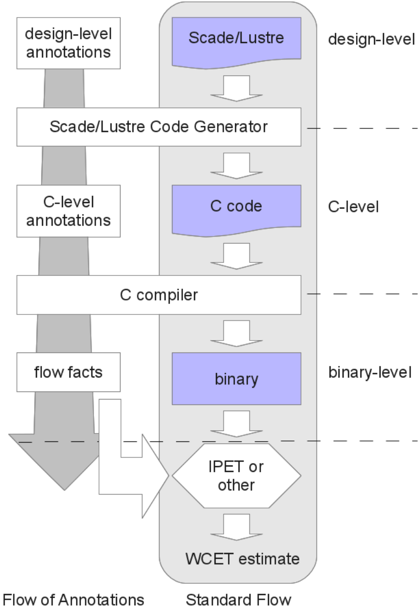


WCET and semantics

- semantics influence the execution path

⇒ "feasible" worst-case path

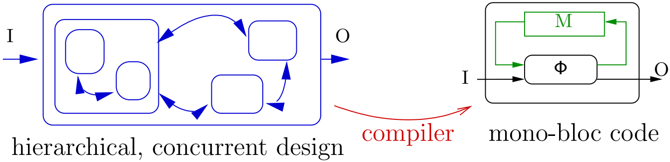
Structure du projet



W-SEPT Partners:

- IRIT Toulouse
- Inria Rennes
- Continental Toulouse
- Verimag Grenoble (coordinator)

Compilation of Synchronous Programs



A basic compiler produces an "object-like" code:

- A (remanent) static memory (no DA).
- A step method: purely sequential code (static scheduling)
  mainly a sequence of assignment + conditional,
  maybe some loops, but statically bounded (e.g. arrays iter/fold)

Remark: providing such a compiler makes the source language a *de facto synchronous language* (e.g. c-code generators for Simulink/Stateflow)

Single Loop Implementation

Basic mono-bloc

- Synchronous compilers produce the application software (the "object"). Implementation depends on a particular middleware/hardware
- Basically, a synchronous code can run "bare metal" (some drivers for input/output, but no complex OS like dynamic scheduler)
- e.g. a simple periodic implementation (period comes form a specialist of the domain):

```
var I, O, M
M := M0
each period do
  read(I)
  O, M = step(I, M)
  write(O)
end
```

Real time?

- Computes an upper *bound* of the WCET of the step method (+ necessary I/O latency)
- check that *bound* $<$ *period*
- If yes: the system does not miss any change from the environment,
  the synchronous hypothesis is valid

Synchronous programming vs Timing Analysis

Open questions

- SP helps TA (at least) because:
  - it guarantees that WCET exists
  - it makes the analysis simpler : almost heap-free code, no aliasing problems
- Could it be better ?
  - Discovering (non-trivial) infeasible path ?

Infeasible path

- A path is semantically impossible because the corresponding conjunction of conditions is always false, for any execution of the program.

⇒ How to make these conditions explicit ?
Find relations between them ?
Check satisfiability ?

  - At binary level/C level: not obvious, require to build some "propositional" representation of the program (e.g. SSA form)
  - At the Lustre level: much more convenient, already in some "clean mathematical" form, moreover verification tools are available.

From code branches to high level

The "theorem"

For any branch in the binary code, there must exist a Boolean expression in the source code whose value coincide to the predicate "the branch is taken".

Proof: otherwise the compiler is certainly buggy!
Problem: OK, but how to find these expressions?

- requires a traceability between binary and C code, C code and Lustre,
- between Lustre and C: not a big problem, the compiler is under control...
- much mode tricky between C and binary (e.g. strongly depends on optimisations)
- The problem has to be investigated ...
- ... Let's try with a simple case.

From code branches to high level

The "theorem"

For any branch in the binary code, there must exist a Boolean expression in the source code whose value coincide to the predicate "the branch is taken".

Proof: otherwise the compiler is certainly buggy!

Problem: OK, but how to find these expressions?

- requires a traceability between binary and C code, C code and Lustre,
- between Lustre and C: not a big problem, the compiler is under control...
- much mode tricky between C and binary (e.g. strongly depends on optimisations)
- The problem has to be investigated ...
- ... Let's try with a simple case.

From code branches to high level

The "theorem"

For any branch in the binary code, there must exist a Boolean expression in the source code whose value coincide to the predicate "the branch is taken".

Proof: otherwise the compiler is certainly buggy!
Problem: OK, but how to find these expressions?

- requires a traceability between binary and C code, C code and Lustre,
- between Lustre and C: not a big problem, the compiler is under control...
- much mode tricky between C and binary (e.g. strongly depends on optimisations)
- The problem has to be investigated ...
- ... Let's try with a simple case.

From code branches to high level

The "theorem"

For any branch in the binary code, there must exist a Boolean expression in the source code whose value coincide to the predicate "the branch is taken".

Proof: otherwise the compiler is certainly buggy!
Problem: OK, but how to find these expressions?

- requires a traceability between binary and C code, C code and Lustre,
- between Lustre and C: not a big problem, the compiler is under control...
- much mode tricky between C and binary (e.g. strongly depends on optimisations)
- The problem has to be investigated ...
- ... Let's try with a simple case.

A first attempt

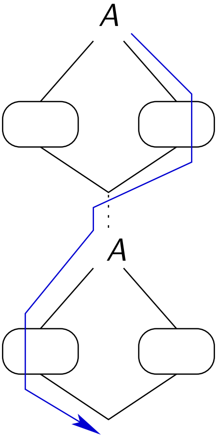
Simple compilation scheme:

- Lustre $\rightarrow$ C compiler adapted in such a way that all conditional are of the form `if (`E`) ...`, where E coincide with a Lustre expression.
- C compiler is called in "debug" mode (no optimization), in such a way that all branching instruction is flagged by the corresponding `if (`E`) ...`
- A path in the binary code corresponds to a conjunction of literals (e.g. $E1 \wedge E2 \wedge \neg E3$)
- A decision tool is used to check if the conjunction (the path) is satisfiable (feasible)

Kinds of infeasibility

Structural/syntactical

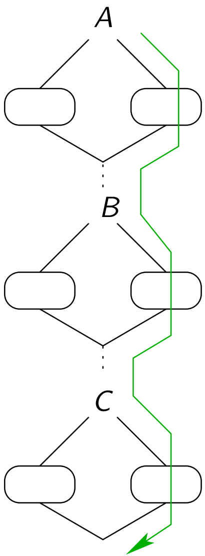
- based on identity
- decision procedure unnecessary
- could have been done at binary level (maybe via SSA reconstruction)



Kinds of infeasibility

Due to statical unsatisfiability

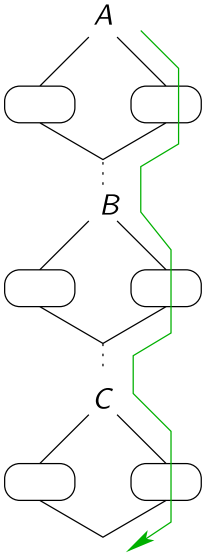
- Examples:
  - `A = E`, `B = not E`,
  - `A = x xor y`, `B = y and z`, `C = x and t`
  - `A = x + y > 10` , `B = y <= 2`, `C = 2*x < 15`
- requires decision procedure, e.g. Sat (Modulo Theory) solver
- could have been done at binary level (maybe via SSA reconstruction)



Kinds of infeasibility

Due to dynamic unreachability

- `A = if PA then not x else (PC and x);`
  `B = PB and not y or PA and x or PC and not x and y;`
  `C = PC and not (y or x) or PC and y;`
- where PA, PB, PC are the values of A, B, C computed at the previous cycle
- (almost) impossible to discover at C/binary level
- strongly depends on the "dynamic" nature of the program
- requires reachability (model-checking) techniques



Conclusion/Questions

Semantic "facts"

- simple ones can be expressed/discovered more easily at the Lustre level ...
- ... as far as we are able to relate binary branches to Lustre expressions !
- complex (dynamic) ones are very hard (impossible) to handle at binary level

Traceability

- dramatically depends on the C compiler
- how to deal with optimisation?

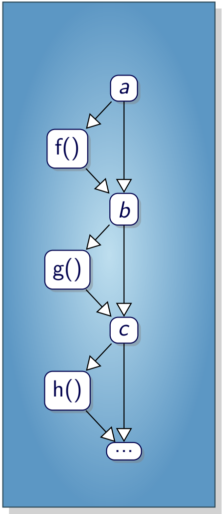
Expressing facts in the WCET estimation

Complexity/Strategy

- checking feasibility can be extremely expensive
- require strategy/heuristics:
  - limitation to pair-wise relations between conditions
  - check satisfiability afterwards (to refute the WCET estimation), or first (to simplify the WCET estimation problem)

- simple pair-wise relations can be expressed in ILP (implications)
- open problem: is it the best method ?

Example



OTAWA: WCET= 3000 cycles

MiniTool: worst-case path in Lustre
a and b and c

Lesar: path is not possible !

lp: $x_f + x_g \leq 1$
$x_f + x_h \leq 1$
$x_g + x_h \leq 1$

lpsolve: WCET= 2500 cycles

Draft of Proof-of-concept

Lustre compiler: variableC
$\Rightarrow$ ExprLustre

C compiler: testC=f(variableC)
$\Rightarrow$ @ instruction

OTAWA: @ instruction $\Rightarrow$ CFG basic block
$\Rightarrow$ worst-case path

MiniTool : worst-case path
$\Rightarrow$ property = conjunction of ExprLustre

Lesar: property
$\Rightarrow$ feasible path?

Open questions

- Traceability from Lustre to C:
  - from C to binary?
  - from Scade to C?
- No loop in our model:
  - only loop bounds?
  - need of loop-iteration identifier?
- Compiler optimization:
  - from not-optimized to fully optimized?
  - from precise estimation of WCET to better code?