Harmonic clocks and how to infer them

Guillaume Iooss, Albert Cohen, Marc Pouzet

ENS - PARKAS

November 30, 2017
Work in progress...
Consider a particular set of clocks (⊂ N-synchronous clocks)

- Strictly periodic
- One activation per period

⇒ \(0^k(10^{n-1})\), \(k\) is the phase and \(n\) the period

Multiple harmonic periods

- Integral ratio between periods
- Synchronized start
Multiperiodic harmonic clocks

- Consider a particular set of clocks ($\subset$ N-synchronous clocks)
  - Strictly periodic
  - One activation per period
  \[ 0^k(10^{n-1}) \], $k$ is the phase and $n$ the period

- Multiple harmonic periods
  - Integral ratio between periods
  - Synchronized start

- Why are we considering this kind of clocks?
  - Frequently used for integration code
  - Property can be exploited to generate efficient code
Related work: Prelude

**Prelude:** language for integration

- Links nodes (implemented in Lustre) together
  - Equational language similar to Lustre
  - Nodes are associated with a wcet
  - No computation outside of node calls

- Force clocks to be of the previously mentioned form
  - Operators to sub/over-sample ($\ast k$ and $/k$)

- Real-time information (duration of a period known)

- Relaxed synchronous hypothesis: execution must end before the next tick of a clock
  $\sim \neq$ Lustre Code Generation strategy (no step function)
Related work: Prelude

**Prelude:** language for integration
- Links nodes (implemented in Lustre) together
  - Equational language similar to Lustre
  - Nodes are associated with a wcet
  - No computation outside of node calls
- Force clocks to be of the previously mentioned form
  - Operators to sub/over-sample ($\times k$ and $/k$)
- Real-time information (duration of a period known)
- Relaxed synchronous hypothesis: execution must end before the next tick of a clock
  - $\sim \neq$ Lustre Code Generation strategy (no step function)
  - Can we do something similar, but in the Lustre formalism?
Determining clocks in an integration specification

- Two components inside a clock: period and a *phase*
  - Issue: need part of the schedule to write the specification
  - Cost function to consider (ex: balancing WCETs)
Determining clocks in an integration specification

- Two components inside a clock: period and a *phase*
  - Issue: need part of the schedule to write the specification
  - Cost function to consider (ex: balancing WCETs)

- **Case study:** Flight control application
  - 6000 nodes, 30k data communicated
  - Base clock is 5 ms
  - Nodes associated to 4 different periods (10/20/40/120 ms)

⇒ Fixing all clocks by hand is tedious
Contributions

1. Formalization of harmonic clocks in Lustre
   - Strict synchronous hypothesis (≠ Prelude)
   - Same expressiveness than Prelude
   - Idea for efficient code generation

2. Clocks partially defined (i.e., period only provided)
   - Infer their phase at compile time
   - Provides minimal information to be deterministic
   - Easier to write multiperiodic Lustre code
Outline

1. Introduction
2. Harmonic clock
3. Partially defined clock
4. Conclusion
Rate tree

- **Rate**: set of strictly periodic clocks sharing the same period
  \[ \{ 0^k(10^{n-1}) \mid 0 \leq k < n \} \]

- **Rate tree**: Tree of rates
  - Root: base rate (contains only the base clock)
  - Edge: represent the harmonic ratio between 2 rates

- **Example**:

  \[ r1 \xrightarrow{2} r2 \xrightarrow{2} r3 \xrightarrow{3} r4 \]
Rate tree

- **Rate**: set of strictly periodic clocks sharing the same period

  \[ \{ 0^k(10^{n-1}) \mid 0 \leq k < n \} \]

- **Rate tree**: Tree of rates
  - Root: base rate (contains only the base clock)
  - Edge: represent the harmonic ratio between 2 rates

- **Example**:

  \[
  \begin{align*}
  r1 & \xrightarrow{2} r2 \xrightarrow{2} r3 \xrightarrow{3} r4
  \end{align*}
  \]

- **Local ratio**: ratio of the incoming edge

- **Global ratio**: product of the ratio of the path from the root
Clock tree

- Edge = sub-clock relation
- Path on this tree = definition of a clock
  Example: $c_4^2 = (1) \text{ on } (10) \text{ on } (01) \text{ on } (100) = 0^2(10^{11})$
- Clock tree derived from rate tree
  → Constructor to build the set of clocks associated to a rate
Navigating the clock tree

Three operators which changes the clock

- **when**: Sub-sampling operator
  
  $$\text{Var2} = \text{Var1 when (FT)};$$
Navigating the clock tree

Three operators which changes the clock

- **when**: Sub-sampling operator
  \[ \text{Var2} = \text{Var1} \text{ when (FT)}; \]

- **merge**: Can use it to over-sample
  \[ \text{Var2} = \text{merge (FT) Var1 (Init fby Var2)}; \]
  where \( \text{Var1} :: c2^1, \text{Var2} :: c1 \)
Navigating the clock tree

Three operators which changes the clock

- **when:** Sub-sampling operator
  \[ \text{Var2} = \text{Var1} \text{ when (FT);} \]

- **merge:** Can use it to over-sample
  \[ \text{Var2} = \text{merge (FT)} \text{ Var1 (Init fby Var2);} \]
  where \(\text{Var1} :: \text{c2}^1, \text{Var2} :: \text{c}1\)
  ⇒ Syntactic sugar: current (over-sampling operator)
  \[ \text{Var2} = \text{current(c2}^1, \text{c}1, \text{Init, Var1}); \]
Navigating the clock tree

Three operators which changes the clock

- **when**: Sub-sampling operator
  \[ \text{Var2} = \text{Var1} \text{ when (FT)}; \]

- **merge**: Can use it to over-sample
  \[ \text{Var2} = \text{merge (FT) Var1 (Init fby Var2)}; \]
  where \( \text{Var1} :: c_2^1, \text{Var2} :: c_1 \)
  \[ \Rightarrow \text{Syntactic sugar: current (over-sampling operator)} \]
  \[ \text{Var2} = \text{current}(c_2^1, c_1, \text{Init}, \text{Var1}); \]

- **buffer1**: Communicate between clocks of the same rate
  \[ \text{Var2} = \text{buffer1}(c_3^1, c_3^3, \text{Init}, \text{Var1}); \]
  where \( \text{Var1} :: c_3^1, \text{Var2} :: c_3^3 \)
  \[ \Rightarrow \text{Can be viewed as combination of when/merge} \]
Efficient Code Generation

- Step function $\leftrightarrow$ base clock
- Using these operators allows us to compile them efficiently

- buffer1
  - Interleaving between clocks known $\sim$ buffer of size 1
  - Use a set and a get
Efficient Code Generation

- Step function $\leftrightarrow$ base clock
- Using these operator allows us to compile them efficiently

- buffer1
  - Interleaving between clocks known $\sim$ buffer of size 1
  - Use a set and a get

- current
  - One update which is repeated until the next one
  - Use a set for multiple get
Outline

1. Introduction
2. Harmonic clock
3. Partially defined clock
4. Conclusion
Partially defined clocks

- Phase of a clock = part of a (large-grain) schedule
  - Have to take into account complicated criterion (wcet balancing, freshness, memory usage, ...)
  - Choice reflected in the equations (correct clocking)
Partially defined clocks

- Phase of a clock = part of a (large-grain) schedule
  - Have to take into account complicated criterion (wcet balancing, freshness, memory usage, ...)
  - Choice reflected in the equations (correct clocking)

- **Issue:** need to specify it in the Lustre code
  - Tedious for large applications
Partially defined clocks

- Phase of a clock = part of a (large-grain) schedule
  - Have to take into account complicated criterion (wcet balancing, freshness, memory usage, ...)
  - Choice reflected in the equations (correct clocking)

- **Issue**: need to specify it in the Lustre code
  - Tedious for large applications

- **Solution**: Only specify the period for some variable.
  - Compiler find automatically the phase
Non-determinism

- Variable declaration: can specify only the rate of a clock
  \[ \text{Var::rate(r1)} \]
Non-determinism

- Variable declaration: can specify only the rate of a clock
  \[ \text{Var::rate}(r_1) \]
- Implicit buffer surrounding each use of this variable
  - Data available to all phases after it is produced
  - Buffer will be explicit once the phase is determined
Non-determinism

- Variable declaration: can specify only the rate of a clock
  \[ \text{Var::rate}(r1) \]
- Implicit buffer surrounding each use of this variable
  - Data available to all phases after it is produced
  - Buffer will be explicit once the phase is determined

- **Issue:** non-determinism on dependences with different rates
Determinism - fast to slow rate

- Need to precise which value of Var1 is used
Determinism - fast to slow rate

- Need to precise which value of Var1 is used
- **Idea:** New boolean variable \( b_{when}(r, i) \), \( 0 \leq i < \text{locRatio}(r) \)
  \[
  \text{Var2} = f(\text{Var1 where } b_{when}(r2, 1))
  \]
  where \( b_{when}(r2, 1) \):
  - Is a fresh variable
  - Has the same clock than Var1 (unknown right now)
  - \( b_{when}(r2, 1) = (\text{FTF}) \)
- \( b_{when} \) can automatically be obtained with the rate tree
Determinism - slow to fast rate

Need to precise when the value of Var2 is available
Need to precise when the value of Var2 is available

**Idea:** reuse the previously introduced boolean $b_{\text{when}}(r,i)$

$$Var1 = f(\text{merge } b_{\text{when}}(r2,2) \ Var2$$

$$\quad \quad (0 fby Var1));$$

where $b_{\text{when}}(r2,2) :: \text{clock}(Var1) = (\text{FFT})$ will be a fresh variable
Constraint extraction

We want to find the phase of all variable with incomplete information

- \( p_{Var} \): phase of variable \( Var :: rate(r_{Var}) \)

- Bounds: \( 0 \leq p_{Var} < global\_ratio(r_{Var}) \)
Constraint extraction

We want to find the phase of all variables with incomplete information:

- $p_{Var}$: phase of variable $Var :: rate(r_{Var})$

- Bounds: $0 \leq p_{Var} < global\_ratio(r_{Var})$

- For each dependence:
  - End of a producer happens before start of a consumer
    
    $$p_{Prod} + Constant \leq p_{Cons}$$

  - $Constant$: depend on the ratio and element accessed
  - If uses a value from the previous cycle (ex: fby), no constraint
Constraint extraction

We want to find the phase of all variable with incomplete information

- $p_{Var}$ : phase of variable $Var :: rate(r_{Var})$

- Bounds: $0 \leq p_{Var} < global\_ratio(r_{Var})$

- For each dependence:
  - End of a producer happens before start of a consumer

\[ p_{Prod} + Constant \leq p_{Cons} \]

- $Constant$: depend on the ratio and element accessed
- If uses a value from the previous cycle (ex: tby), no constraint
- Can add a constraint which forces the use of the previous value before the new value is computed (allow memory reuse)
Constraint extraction (Bonus - 1)

Fast rate to slow rate - Two situations:

- Previous value is used ⇒ No constraint
- $i$-th value used ⇒ $p_{Prod} + i \times k_{base,Prod} \leq p_{Cons}$
Constraint extraction (Bonus - 1)

Fast rate to slow rate - Two situations:
- Previous value is used ⇒ No constraint
- \( i \)-th value used ⇒ \( p_{Prod} + i \times k_{base,Prod} \leq p_{Cons} \)

(Optional constraint) No extra memory
- Previous value is used ⇒ \( p_{Cons} \leq p_{Prod} + 0 \times k_{base,Prod} \)
- \( i \)-th value used ⇒ \( p_{Cons} \leq p_{Prod} + (i + 1) \times k_{base,Prod} \)
Constraint extraction (Bonus - 2)

Slow rate to fast rate - Two situations:

- Prev always used: \((k_{Cons,Prod} - 1) \times k_{base,Cons} + p_{Cons} \leq p_{Prod}\)
- Switch to current val at \(i\)-th: \(p_{Prod} \leq i \times k_{base,Cons} + p_{Cons}\)
Slow rate to fast rate - Two situations:

- Prev always used: \((k_{Cons,Prod} - 1) \times k_{base,Cons} + p_{Cons} \leq p_{Prod}\)
- Switch to current val at \(i\)-th: \(p_{Prod} \leq i \times k_{base,Cons} + p_{Cons}\)

(Optional constraint) No extra memory

- Switch at \(i\)-th value: \(p_{Prod} \leq (i + 1) \times k_{base,Cons} + p_{Cons}\)
Several possibilities (which can be combined). Mostly naive

- Freshness of data
  - Minimize distance between producer and consumer
  - Note: could also be introduced as an extra constraint?
Cost function

Several possibilities (which can be combined). Mostly naive

- Freshness of data
  - Minimize distance between producer and consumer
  - Note: could also be introduced as an extra constraint?

- Memory usage
  - Try to get as many memory reuse as possible
  - Need to introduce a boolean variable per dependence
Several possibilities (which can be combined). Mostly naive

- **Freshness of data**
  - Minimize distance between producer and consumer
  - Note: could also be introduced as an extra constraint?

- **Memory usage**
  - Try to get as many memory reuse as possible
  - Need to introduce a boolean variable per dependence

- **WCET load balancing**
  - Need to introduce a quadratic number of boolean variable (one per equation/possible phase)
  - Quadratic number of constraints added, some of them linear
  ⇒ Costly (how much?)
Solving the ILP

- **Experiment:**
  - Flight control application
    - (6000 nodes, 30k data communicated, 4 different periods)
  - Generate constraints and solve them with glpk
Solving the ILP

- **Experiment:**
  - Flight control application
    (6000 nodes, 30k data communicated, 4 different periods)
  - Generate constraints and solve them with glpk

- **Finding an integral solution:** 1.5 s

- **WCET load balancing cost function:**
  - $\min(A)$ where $\sum \cdots \leq A \Rightarrow$ Best integral solution: $>5$h
    (last solution before stopping: 200 cycles more than rational)
Solving the ILP

- **Experiment:**
  - Flight control application
    - (6000 nodes, 30k data communicated, 4 different periods)
  - Generate constraints and solve them with glpk

- **Finding an integral solution:** 1.5 s

- **WCET load balancing cost function:**
  - $\min(A)$ where $\sum \cdots \leq A \Rightarrow$ Best integral solution: ??? > 5h
    - (last solution before stopping: 200 cycles more than rational)
  - $\sum \cdots \leq 1.01 \times \text{rat}_\text{sol} : \approx 32$ min
  - $\sum \cdots \leq 1.25 \times \text{rat}_\text{sol} : \approx 28$ min
  - $\sum \cdots \leq 1.5 \times \text{rat}_\text{sol} : \approx 11.3$ min

- Last part: reinject a solution in the Lustre program
  - Clocks are now fully defined
  - Explicit buffer when needed
  - Can verify the validity of the solution

⇒ Ends up with a classical Lustre program
Solving the ILP

- **Experiment:**
  - Flight control application
    - (6000 nodes, 30k data communicated, 4 different periods)
  - Generate constraints and solve them with glpk

- **Finding an integral solution:** 1.5 s

- **WCET load balancing cost function:**
  - $\min(A) \text{ where } \sum \cdots \leq A \Rightarrow \text{Best integral solution: } ??? > 5h$
  - (last solution before stopping: 200 cycles more than rational)
  - $\sum \cdots \leq 1.01 \ast \text{rat}_sol : \approx 32 \text{ min}$
  - $\sum \cdots \leq 1.25 \ast \text{rat}_sol : \approx 28 \text{ min}$
  - $\sum \cdots \leq 1.5 \ast \text{rat}_sol : \approx 11.3 \text{ min}$

- **Last part:** reinject a solution in the Lustre program
  - Clocks are now fully defined
  - Explicit buffer when needed
  - Can verify the validity of the solution

$\Rightarrow$ Ends up with a classical Lustre program
Outline

1. Introduction
2. Harmonic clock
3. Partially defined clock
4. Conclusion
Conclusion

- Formalism of harmonic clocks in Lustre
- Extension to specify only the period of a clock
Conclusion

- Formalism of harmonic clocks in Lustre
- Extension to specify only the period of a clock

**Current/future work:**

- Implementation in Heptagon
- Can add duration to nodes (long tasks)
- Cost function to be improved:
  - How to group nodes for parallelism?
- Natural extension: non-determinism

  \[ \text{Var2} = f(\text{Var1 when rate(r2)}); \]

  \[ \Rightarrow \text{ILP: remove the corresponding constraint} \]
Formalism of harmonic clocks in Lustre

Extension to specify only the period of a clock

Current/future work:
- Implementation in Heptagon
- Can add duration to nodes (long tasks)
- Cost function to be improved:
  - How to group nodes for parallelism?
- Natural extension: non-determinism
  \[ \text{Var2} = f(\text{Var1 when rate(r2)}) \]
  \[ \Rightarrow \text{ILP: remove the corresponding constraint} \]
  \[ \approx \text{Hyperperiod extension (→ Prelude?)} \]