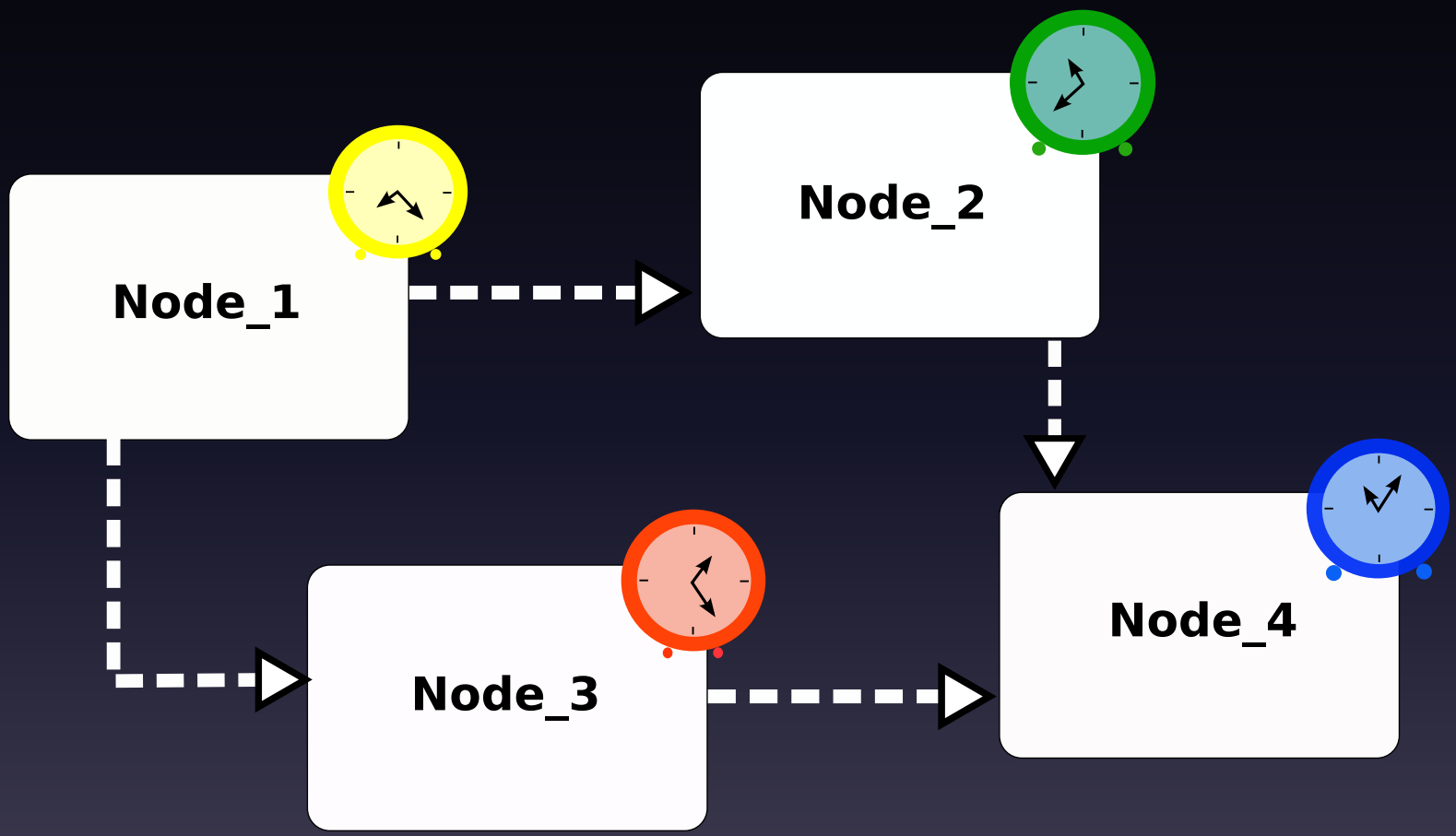
Static Analysis of Signal Programs for Efficient Design of Multi-Clocked Embedded Systems

Abdoulaye Gamatié Laure Gonnord

CNRS/LIFL and INRIA, France

April 2011

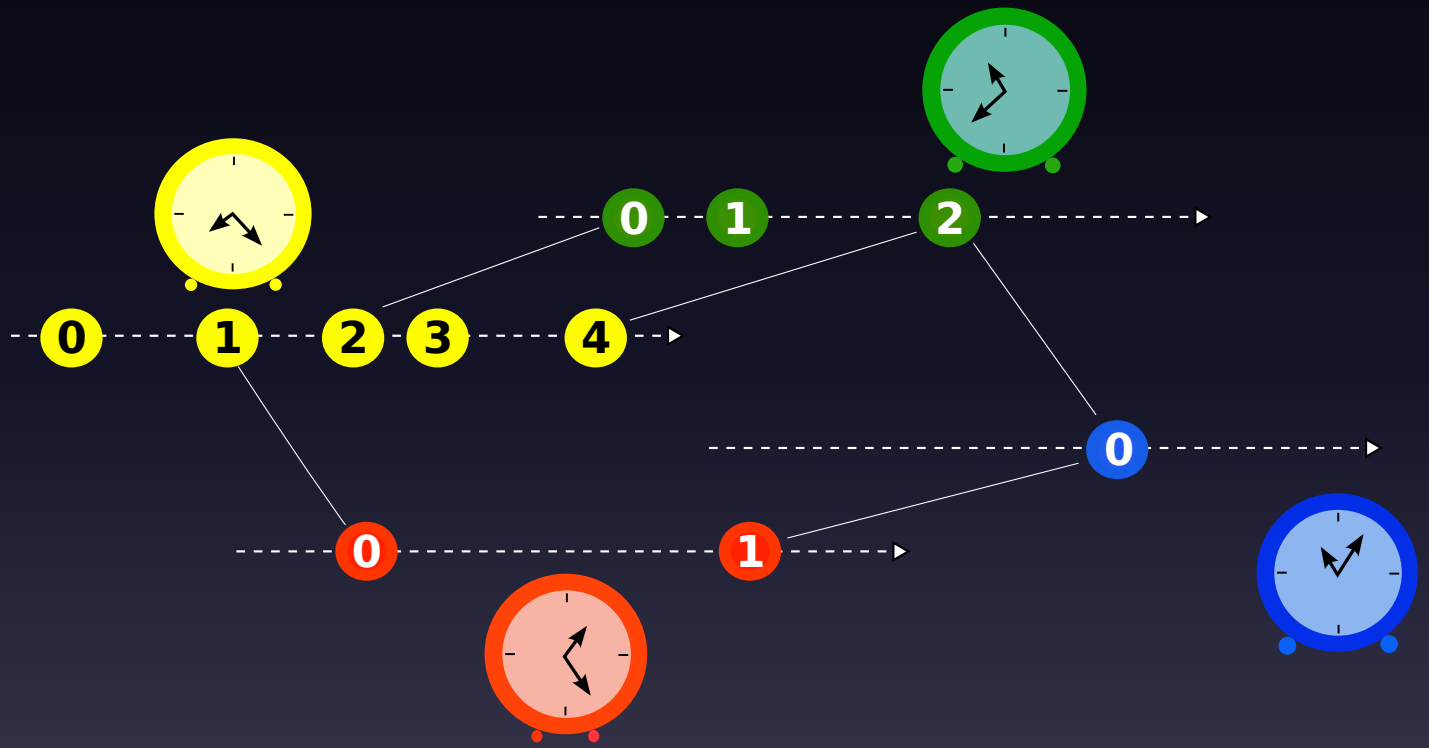
Multi-clocked embedded systems



Examples: GALS systems, MPSoCs

Multi-clocked system modeling

Node interactions specified using abstract clock relations



Examples: Clock Constraint Specification Language (CCSL), Multi-Rate Instantaneous Channel connected Data Flow (MRICDF), Signal

In this talk...

1. Synchronous language Signal
   - main concepts
   - compilation: static analysis & code generation
2. A solution for improving the compilation
   - a new abstraction for programs
   - illustration and implementation
3. Concluding remarks

Signal language

Basic notions

- signal x: sequence $(x_{t_i})_{t_i \in \mathbb{N}}$ of typed values ($\perp$ is absence)
- clock $\widehat{\mathrm{x}}$ of a signal x: instants where values $\neq \perp$
- process $\mathrm{y} := \mathrm{x}+1$: relations between values/clocks of signals

time	:	t_0	t_1	t_2	t_3	t_4	t_5	$\ldots$
x	:	1	5	$\perp$	6	$\perp$	0	$\ldots$
$\widehat{\mathrm{x}}$	:	*tt*	*tt*	*ff*	*tt*	*ff*	*tt*	$\ldots$
y	:	2	6	$\perp$	7	$\perp$	1	$\ldots$

Primitive operators on signals

Synchronous operators: signals have the same clock

- Relations. $y := f(x_1, ..., x_n)$

  Example: level := pre_level - pump

pre_level	:	$\perp$	0.3	$\perp$	$\perp$	-2.7	5	1	...
pump	:	$\perp$	3	$\perp$	$\perp$	-7.7	4	0.5	...
level	:	$\perp$	-2.7	$\perp$	$\perp$	5	1	0.5	...

Primitive operators on signals

Synchronous operators: signals have the same clock

- Relations. $y := f(x_1, ..., x_n)$

  Example: level := pre_level - pump

pre_level	:	$\perp$	0.3	$\perp$	$\perp$	-2.7	5	1	...
pump	:	$\perp$	3	$\perp$	$\perp$	-7.7	4	0.5	...
level	:	$\perp$	-2.7	$\perp$	$\perp$	5	1	0.5	...

- Delay. $y := x \$ 1 \text{ init } c$

  Example: pre_level := level $ 1 init 0.3

level	:	$\perp$	-2.7	$\perp$	$\perp$	5	1	0.5	...
pre_level	:	$\perp$	0.3	$\perp$	$\perp$	-2.7	5	1	...

Primitive operators on signals

Multi-clock operators: signals may have different clocks

- Sampling. y := x when b

  Example: alarm := empty when (0 >= level)

empty	:	5	$\perp$	4	8	7	3	$\perp$	...
level	:	-1	5	$\perp$	3	-9	$\perp$	$\perp$	...
(0 >=level)	:	*tt*	*ff*	$\perp$	*ff*	*tt*	$\perp$	$\perp$	...
alarm	:	5	$\perp$	$\perp$	$\perp$	7	$\perp$	$\perp$	...

Primitive operators on signals

Multi-clock operators: signals may have different clocks

- Sampling. y := x when b

  Example: alarm := empty when (0 >= level)

empty	:	5	$\perp$	4	8	7	3	$\perp$	...
level	:	-1	5	$\perp$	3	-9	$\perp$	$\perp$	...
(0 >=level)	:	*tt*	*ff*	$\perp$	*ff*	*tt*	$\perp$	$\perp$	...
alarm	:	5	$\perp$	$\perp$	$\perp$	7	$\perp$	$\perp$	...

- Merging. z := x default y

  Example: global_alarm := scarce default overflow

scarce	:	*ff*	$\perp$	*tt*	*tt*	$\perp$	$\perp$	$\perp$	...
overflow	:	*tt*	*tt*	$\perp$	*ff*	$\perp$	*ff*	$\perp$	...
global_alarm	:	*ff*	*tt*	*tt*	*tt*	$\perp$	*tt*	$\perp$	...

Primitive operators on processes

- Composition: $\mathrm{P}_1 \mid \mathrm{P}_2$

  Example: (| scarce := (0 >= level) | alarm := empty when scarce |)

empty	:	5	$\perp$	4	8	7	3	$\perp$	...
level	:	-1	5	$\perp$	3	-9	$\perp$	$\perp$	...
scarce	:	*tt*	*ff*	$\perp$	*ff*	*tt*	$\perp$	$\perp$	...
alarm	:	5	$\perp$	$\perp$	$\perp$	7	$\perp$	$\perp$	...

Primitive operators on processes

- Composition: $\mathrm{P}_1 \mid \mathrm{P}_2$

  Example: (| scarce := (0 >= level) | alarm := empty when scarce |)

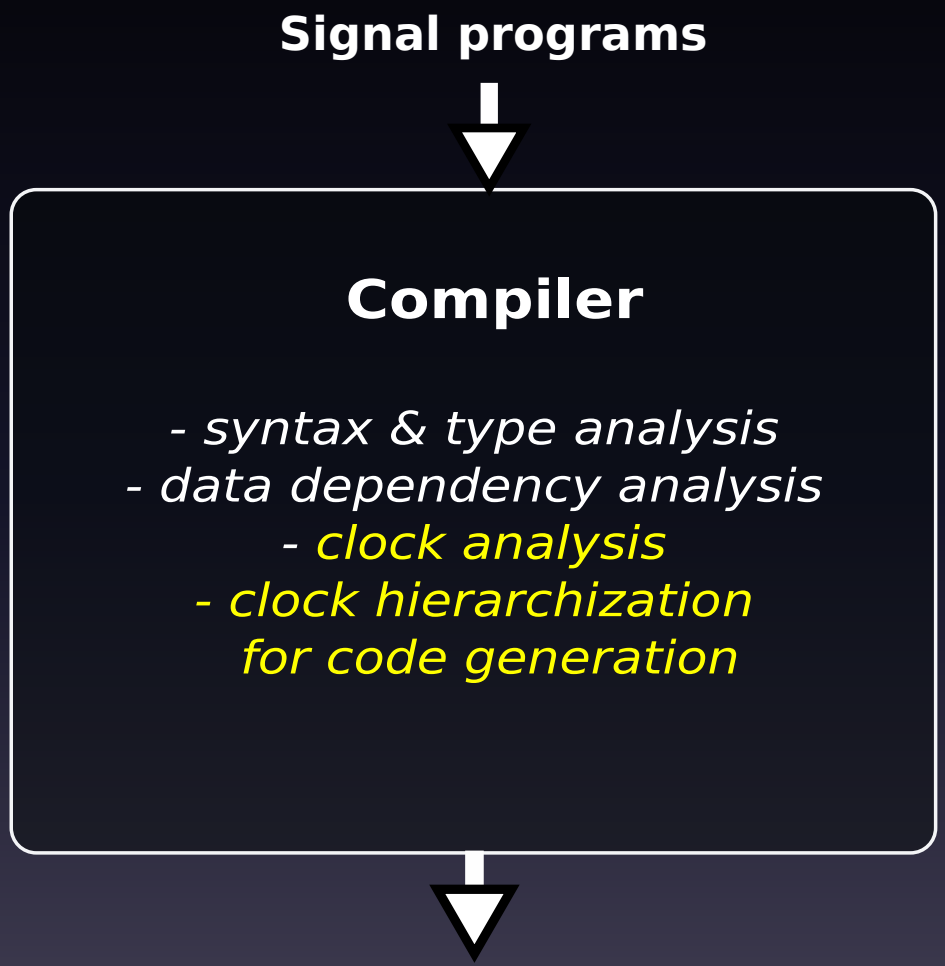
empty	:	5	$\perp$	4	8	7	3	$\perp$	...
level	:	-1	5	$\perp$	3	-9	$\perp$	$\perp$	...
scarce	:	*tt*	*ff*	$\perp$	*ff*	*tt*	$\perp$	$\perp$	...
alarm	:	5	$\perp$	$\perp$	$\perp$	7	$\perp$	$\perp$	...

- Local declaration: P where u

  Example: (| scarce := (0 >= level) | alarm := empty when scarce |) where scarce

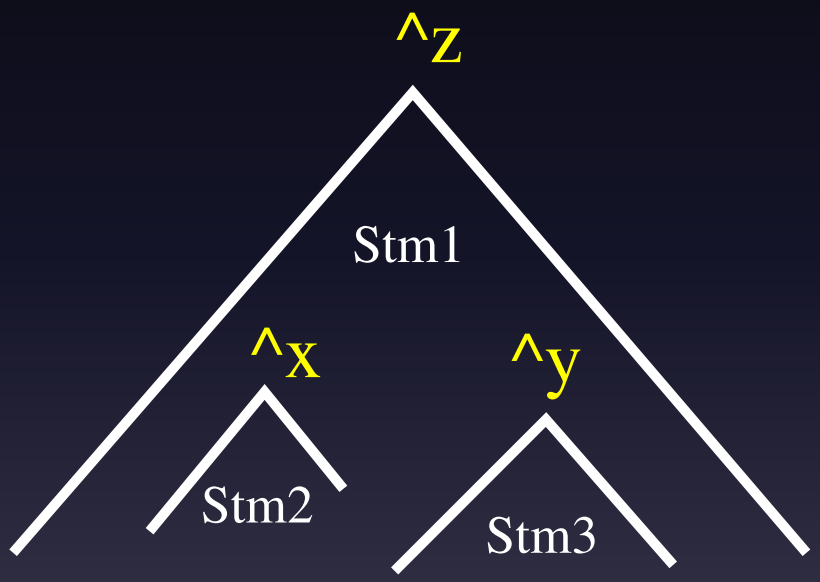
empty	:	5	$\perp$	4	8	7	3	$\perp$	...
level	:	-1	5	$\perp$	3	-9	$\perp$	$\perp$	...
alarm	:	5	$\perp$	$\perp$	$\perp$	7	$\perp$	$\perp$	...

Compilation of Signal programs



Compilation of Signal programs (2)

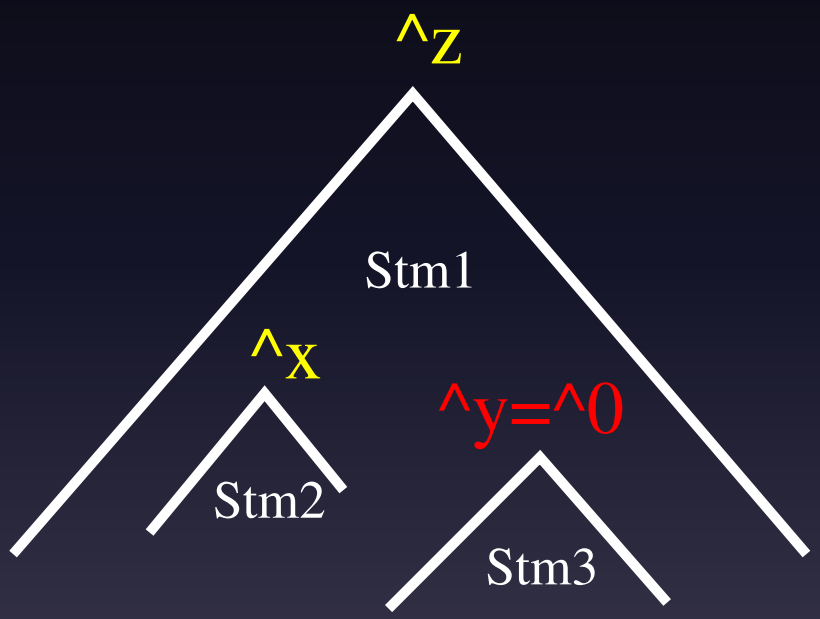
Hierarchization of clocks for code generation




```
if (clk_z)
  { Stm1;
    if (clk_x)
      { Stm2; };
    if (clk_y)
      { Stm3; };
  }
```

Compilation of Signal programs (3)

Analysis of clock constraints for better results




```
if (clk_z)
  { Stm1;
    if (clk_x)
      { Stm2; };
    if (clk_y)
      { Stm3; };
  }
```

Compilation of Signal programs (4)

Boolean abstraction for clock analysis

Example:

```
(| scarce := (0 >= level)
 | overflow := (level > 7)
 | alarm := (true when scarce) when overflow
 |)
```

$$\begin{cases} \hat{}\text{scarce} \Leftrightarrow \hat{}\text{level} \\ \text{scarce} \Leftrightarrow (0 >= level) \end{cases}$$

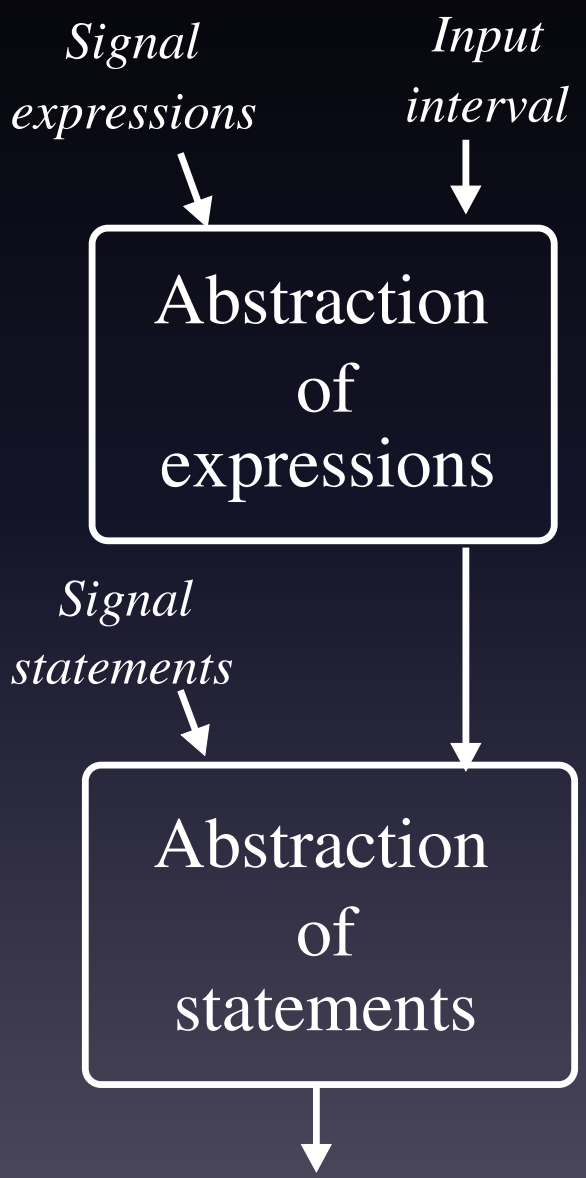
$$\begin{cases} \hat{}\text{overflow} \Leftrightarrow \hat{}\text{level} \\ \text{overflow} \Leftrightarrow (level > 7) \end{cases}$$

$$\begin{cases} \hat{}\text{alarm} \Leftrightarrow [\text{scarce}] \wedge [\text{overflow}] \\ \hat{}\text{scarce} \Leftrightarrow [\text{scarce}] \vee [\neg\text{scarce}] \\ false \Leftrightarrow [\text{scarce}] \wedge [\neg\text{scarce}] \\ \hat{}\text{overflow} \Leftrightarrow [\text{overflow}] \vee [\neg\text{overflow}] \\ false \Leftrightarrow [\text{overflow}] \wedge [\neg\text{overflow}] \\ \text{alarm} \Leftrightarrow [\text{scarce}] \wedge [\text{overflow}] \end{cases}$$

Numerical expressions not fully addressed in abstraction:

$\hat{}\text{alarm} \Leftrightarrow false$ is not detected!

A new abstraction for Signal



given variation intervals of input signals $x_i \in X_{\mathrm{P}}$ of a process P

$\phi(\text{b1 and b2}) = b_1 \wedge b_2$
$\phi(\text{e1 + e2}) = I_{e_1} \widetilde{+} I_{e_2}$
...

first order logic formula $\Phi(\mathrm{P})$ as a set of valuations $v = (\widehat{\ }, \widetilde{\ })$:

$$\begin{array}{lll} \widehat{\ }: & X_{\mathrm{P}} \to \{true, false\} & (clock) \\ \widetilde{\ }: & X_{\mathrm{P}} \to \mathbb{R} \cup \{true, false\} & (value) \end{array}$$

Abstraction of statement behaviors

Examples (y is numeric)

- Relations

$$\Phi(\text{y:=f(x1,...xn)}) = \bigwedge_{i=1}^{n} (\widehat{y} \Leftrightarrow \widehat{x_i}) \wedge \left(\widehat{y} \Rightarrow \widetilde{y} \in \phi(\text{f(x1,...xn)})\right)$$

$$\{v \mid \quad v = \langle \text{x1} \mapsto (\textit{ff}, x_{11}), ..., \text{xn} \mapsto (\textit{ff}, x_{n1}), \text{y} \mapsto (\textit{ff}, y_1)\rangle \ \ or$$
$$v = \langle \text{x1} \mapsto (\textit{tt}, x_{11} \in I_{\text{x1}}), ..., \text{xn} \mapsto (\textit{tt}, x_{n1} \in I_{\text{xn}}), \text{y} \mapsto (\textit{tt}, y_1 \in \widetilde{f}(I_{\text{x1}}...))\rangle\}$$

- Sampling

$$\Phi(\text{y:= x when b}) = \left(\widehat{y} \Leftrightarrow (\widehat{x} \wedge \widehat{b} \wedge \widetilde{b})\right) \bigwedge \left(\widehat{y} \Rightarrow \widetilde{y} = \widetilde{x}\right)$$

- Composition

$$\Phi(\text{P}_1|\text{P}_2) = \Phi(\text{P}_1) \wedge \Phi(\text{P}_2)$$

Concretization of a formula Φ

- Set of events according to all valuations $v \models \Phi$:

  E.g., $v = \langle \text{x1} \mapsto (true, x_{11}), \text{x2} \mapsto (false, x_{21}) ... \text{xn} \mapsto (true, x_{n1}) \rangle$

		v	v'	v''
		$\Downarrow$	$\Downarrow$	$\Downarrow$
x1	:	x_{11}	$\bot$	x_{12}
x2	:	$\bot$	x_{21}	x_{22}
...				
xn	:	x_{n1}	x_{n2}	x_{n3}

Concretization of a formula Φ (2)

- $\Gamma(\Phi) =$ set of all possible traces built from valid events:

$$T_1 = \begin{array}{|llllll|} \hline \text{x1} & : & \bot & \ldots & x_{11} & \ldots \\ \text{x2} & : & x_{21} & \ldots & \bot & \ldots \\ \ldots & & & & & \ldots \\ \text{xn} & : & v_{n2} & \ldots & v_{n1} & \ldots \\ \hline \end{array}$$

$$T_2 = \begin{array}{|llllll|} \hline \text{x1} & : & x_{11} & x_{12} & \bot & \ldots \\ \text{x2} & : & \bot & x_{22} & x_{21} & \ldots \\ \ldots & & & & & \ldots \\ \text{xn} & : & x_{n1} & x_{n3} & x_{n2} & \ldots \\ \hline \end{array}$$

Abstraction soundness

Proposition: *Given a process* P[1] *and a formula* φ*,*

$$\textit{if } \Phi_P \Rightarrow \varphi \textit{, then } [\![\mathrm{P}]\!] \subseteq \Gamma(\varphi) \qquad -- \quad \mathrm{P} \textit{ satisfies } \varphi$$

Yices SMT solver is used to check $\Phi_P \Rightarrow \varphi$

[1] $[\![\mathrm{P}]\!]$ denotes the set of all possible traces satisfying P.

A simple example

Example 2 (cont'd): let program P be:

```
(| scarce := (0 >= level)
 | overflow := (level > 7)
 | alarm := (true when scarce) when overflow
 |)
```

To verify that alarm never occurs when P executes, we consider

1. the abstraction $\Phi(\mathrm{P})$ yields

$$\Phi_P = \begin{cases} (\hat{}\mathrm{overflow} \Leftrightarrow \hat{}\mathrm{level} \Leftrightarrow \hat{}\mathrm{scarce}) \\ \wedge\ (\tilde{}\mathrm{scarce} \Leftrightarrow (\mathrm{level} \in]-\infty, 0])) \\ \wedge\ (\tilde{}\mathrm{overflow} \Leftrightarrow (\mathrm{level} \in]7, +\infty[)) \\ \ldots \end{cases}$$

2. a property φ of interest

$$\varphi = \neg\big((\tilde{}\mathrm{scarce} \wedge \hat{}\mathrm{scarce}) \wedge (\tilde{}\mathrm{overflow} \wedge \hat{}\mathrm{overflow})\big)$$

A simple example (2)

Since $\Phi(\mathrm{P})$ implies φ, we define $\mathrm{P'} = \mathrm{P} \mid \Gamma(\varphi)$ as:

```
(| (| scarce := (0 >= level)
   | overflow := (level > 7)
   | alarm := true when scarce when overflow
   |)
 | (| true when scarce when overflow ^= ^0 |)
 |)
```

Thanks to [jcsc03], $[\![\mathrm{P'}]\!] = [\![\mathrm{P}]\!]$: numerical properties abstracted away by Boolean abstraction of P are made explicit in $\mathrm{P'}$ now!

Another example: Bathtub

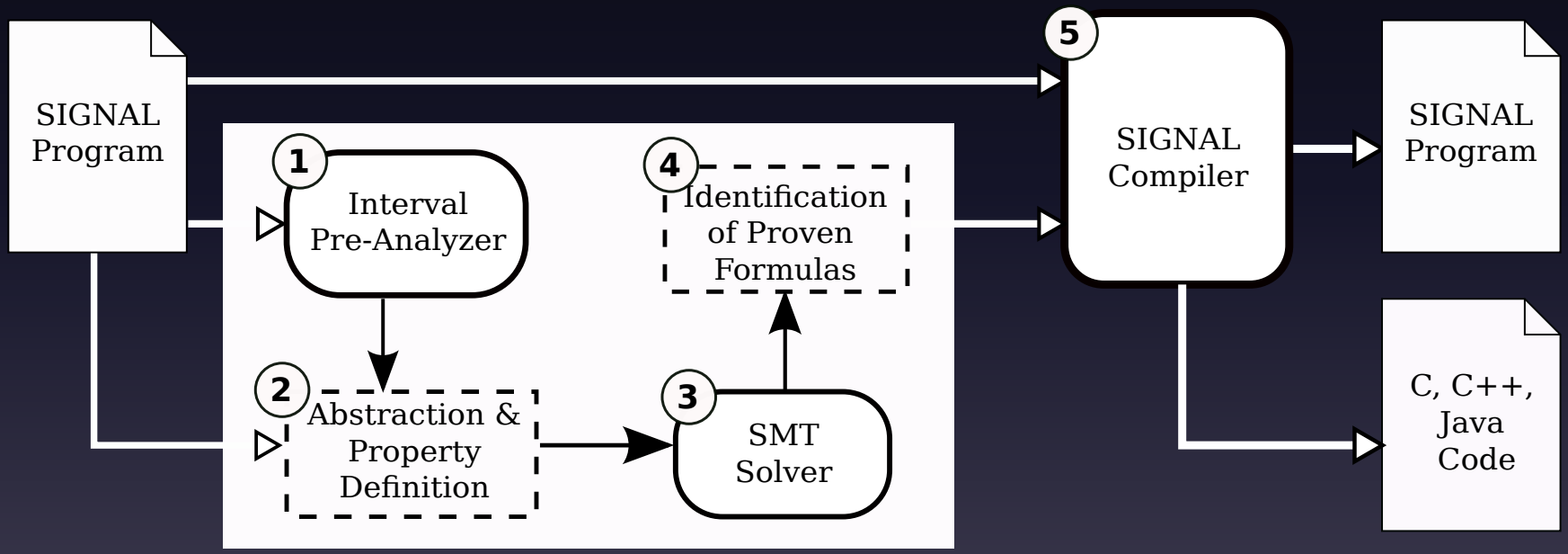
Description and analysis of a Bathtub (see paper)

- process Bathtub: 18 lines in Signal
- Identified clock properties: P1, P2, P3

Resulting clock analysis and code generation

process	*clock issues*	*C code (#lines)*
Bathtub	clock constraints	88
Bathtub | P1	one null clock solved	78
Bathtub | P1 | P2 | P3	five null clocks solved	35

Overview of the solution



Concluding remarks

Improvement of static analysis for multi-clock designs in Signal

- an expressive abstraction (Boolean and numeric parts of programs)
- efficient clock consistency analysis
- optimized automatic code generation
- related works: SAT [fac04] and IDD [ecbs08] for Signal, SMT [fmcad08] for Lustre, Polyhedra abstract inter. [sas99] for synchronous languages

Next steps...

- benchmarks and integration in Signal design environment (Polychrony)
- Signal encoding of formula concretizations

Thanks!

In this talk...

1. Synchronous language Signal
   - main concepts
   - compilation: static analysis & code generation
2. A solution for improving the compilation
   - a new abstraction for programs
   - illustration and implementation
3. Concluding remarks