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

work published at LCTES 2011

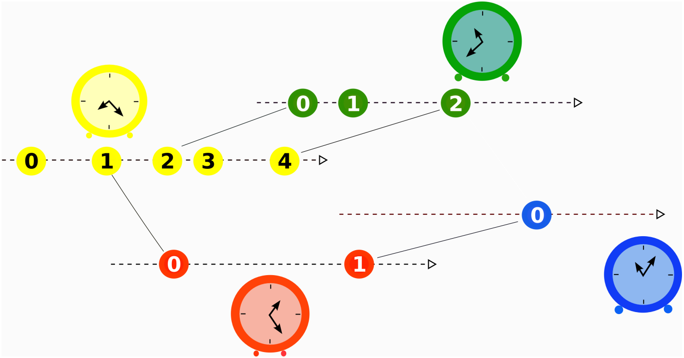
Abdoulaye Gamatié and Laure Gonnord

LIFL/CNRS/INRIA

Synchron, November 2011

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 $\widehat{\mathtt{x}}$ of a signal `x` : instants where values $\neq \perp$
- process `y:= 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	$\ldots$
$\widehat{\mathtt{x}}$	:	*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.** $\texttt{y := f(x}_1\texttt{,..., x}_n\texttt{)}$

  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 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	$\perp$	$\ldots$
`level`	:	-1	5	$\perp$	3	-9	$\perp$	$\perp$	$\ldots$
`(0 >=level)`	:	*tt*	*ff*	$\perp$	*ff*	*tt*	$\perp$	$\perp$	$\ldots$
`alarm`	:	5	$\perp$	$\perp$	$\perp$	7	$\perp$	$\perp$	$\ldots$

- Merging. `z := x default y`

  Example : `global_alarm := scarce default overflow`

`scarce`	:	*ff*	$\perp$	*tt*	*tt*	$\perp$	$\perp$	$\perp$	$\ldots$
`overflow`	:	*tt*	*tt*	$\perp$	*ff*	$\perp$	*ff*	$\perp$	$\ldots$
`global_alarm`	:	*ff*	*tt*	*tt*	*tt*	$\perp$	*ff*	$\perp$	$\ldots$

Primitive operators on processes

- Composition : `P`$_1$ `|` `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 : `P`$_1$ `|` `P`$_2$

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

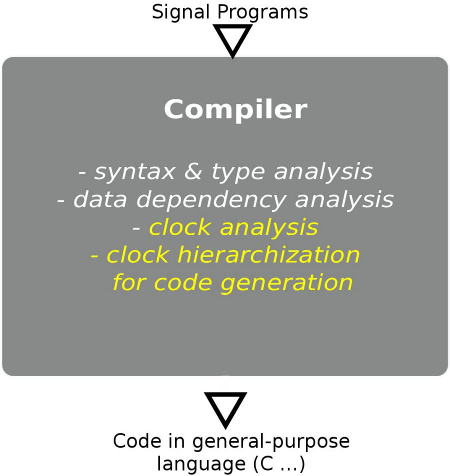
`empty`	:	5	$\perp$	4	8	7	3	$\perp$	...
`level`	:	-1	5	$\perp$	3	-9	$\perp$	$\perp$	...
`scarce`	:	*tt*	*ff*	$\perp$	*ff*	*tt*	$\perp$	$\perp$	...
`alarm`	:	5	$\perp$	$\perp$	$\perp$	7	$\perp$	$\perp$	...

- 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



Signal Programs

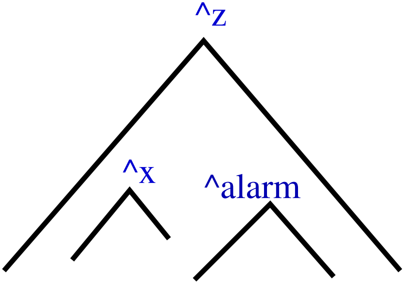
Compiler

- syntax & type analysis
- data dependency analysis
- clock analysis
- clock hierarchization for code generation

Code in general-purpose language (C ...)

Compilation of Signal programs (2)

Clock hierarchy and code generation, improving the final result :



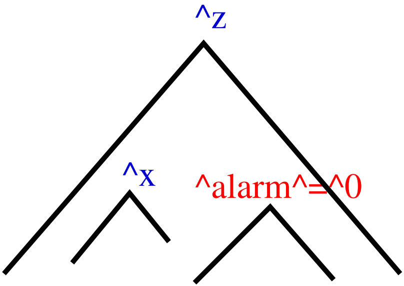
```
if (clk_z) {
   Stm1;
    if (clk_x) {
           Stm2;
       };

if (clk_alarm)   {
            Stm3;
       };

}
```

Compilation of Signal programs (2)

Clock hierarchy and code generation, improving the final result :



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

Compilation of Signal programs (3)

Boolean abstraction for clock analysis

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

$$\begin{cases} \hat{\texttt{scarce}} \Leftrightarrow \hat{\texttt{level}} \\ \texttt{scarce} \Leftrightarrow (0 \geq level) \end{cases}$$

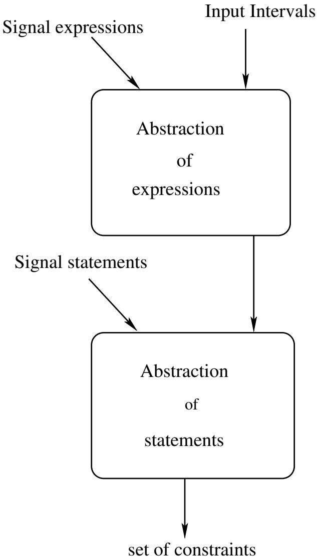
$$\begin{cases} \hat{\texttt{overflow}} \Leftrightarrow \hat{\texttt{level}} \\ \texttt{overflow} \Leftrightarrow (level > 7) \end{cases}$$

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

Numerical expressions not fully addressed in abstraction :

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

A new abstraction for Signal



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

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

...

first order logic formula $\Phi(\mathtt{P})$ as a set of valuations $v = (\hat{\ }, \tilde{\ })$:

$$\begin{array}{lll} \hat{\ } : & X_{\mathtt{P}} \rightarrow \{true, false\} & (clock) \\ \tilde{\ } : & X_{\mathtt{P}} \rightarrow \mathbb{R} \cup \{true, false\} & (value) \end{array}$$

Abstraction of statement behaviors - 1/2

(y is numeric, similar rules for booleans.)

- Relations

$$\Phi(\mathtt{y:=f(x1,\ldots xn)}) = \bigwedge_{i=1}^{n} (\widehat{y} \Leftrightarrow \widehat{x}_i) \wedge \Big(\widehat{y} \Rightarrow \widetilde{y} \in \phi(\mathtt{f(x1,\ldots xn)})\Big)$$

$$\begin{aligned}\{v \mid \quad & v = \langle \mathtt{x1} \mapsto (\mathit{ff}, x_{11}), \ldots, \mathtt{xn} \mapsto (\mathit{ff}, x_{n1}), \mathtt{y} \mapsto (\mathit{ff}, y_1)\rangle \;\; \mathit{or} \\ & v = \langle \mathtt{x1} \mapsto (\mathit{tt}, x_{11} \in I_{\mathtt{x1}}), \ldots, \mathtt{xn} \mapsto (\mathit{tt}, x_{n1} \in I_{\mathtt{xn}}), \mathtt{y} \mapsto (\mathit{tt}, y_1 \in \widetilde{f}(I_{\mathtt{x1}} \ldots))\rangle\}\end{aligned}$$

- Delay

$$\Phi(\mathtt{y:=\ x\ \$\ 1\ init\ c}) = (\widehat{y} \Leftrightarrow \widehat{x}) \bigwedge (\widehat{y} \Rightarrow (\widetilde{y} = \widetilde{x} \vee \widetilde{y} = c))$$

- Sampling

$$\Phi(\mathtt{y:=\ x\ when\ b}) = \Big(\widehat{y} \Leftrightarrow (\widehat{x} \wedge \widehat{b} \wedge \widetilde{b})\Big) \wedge \Big(\widehat{y} \Rightarrow \widetilde{y} = \widetilde{x}\Big)$$

Abstraction of statement behaviors - 2/2

- Merging

$$\Phi(\texttt{y:= x default z}) = \left(\widehat{y} \Leftrightarrow (\widehat{x} \vee \widehat{z})\right) \wedge$$

$$\left(\widehat{y} \Rightarrow ((\widehat{x} \wedge (\widetilde{y} = \widetilde{x})) \vee (\neg\widehat{x} \wedge (\widetilde{y} = \widetilde{z})))\right)$$

- Local signals

$$\Phi(\texttt{P where x}) = \exists\widetilde{x}, \exists\widehat{x} \,.\, \Phi_P$$

- Composition

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

Concretization of a formula Φ

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

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

$$\begin{array}{lllll}
 & & v & v' & v'' \\
 & & \Downarrow & \Downarrow & \Downarrow \\
\mathtt{x1} & : & x_{11} & \bot & x_{12} \\
\mathtt{x2} & : & \bot & x_{21} & x_{22} \\
\ldots & & & & \\
\mathtt{xn} & : & x_{n1} & x_{n2} & x_{n3}
\end{array}$$

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

Abstraction soundness

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

if $\Phi_P \Rightarrow \varphi$, *then* $[\![P]\!] \subseteq \Gamma(\varphi)$ *– –* `P` *satisfies* φ

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

---

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

A simple example

Let `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 $\Phi(\texttt{P})$ yields

$$\left\{\begin{array}{l}
(\hat{\texttt{overflow}} \Leftrightarrow \hat{\texttt{level}} \Leftrightarrow \hat{\texttt{scarce}}) \\
\ \wedge (\tilde{\texttt{scarce}} \Leftrightarrow (\tilde{\texttt{level}} \in]-\infty, 0])) \\
\ \wedge \big(\tilde{\texttt{overflow}} \Leftrightarrow (\tilde{\texttt{level}} \in]7, +\infty[)\big) \\
\hat{\texttt{overflow}})\big) \\
\ \wedge (\hat{\texttt{alarm}} \Leftrightarrow \big(\tilde{\texttt{scarce}} \wedge \hat{\texttt{scarce}} \wedge \tilde{\texttt{overflow}} \wedge \hat{\texttt{overflow}})\big) \\
\ldots
\end{array}\right.$$

2. a property $\varphi = \neg(\hat{alarm})$

A simple example (2)

Recall $\varphi = \neg(\hat{}alarm)$.

- Yices gives : $\Phi(\texttt{P}) \Rightarrow \varphi$.
- (def of Γ) All traces in $\Gamma(\varphi)$ satisfy :

$$\forall t, alarm_t = \bot$$

▶ $[\![\texttt{P}]\!] \subseteq [\![$ `^alarm ^= ^0` $]\!]$

Let P' be :

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

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

Another example : Bathtub

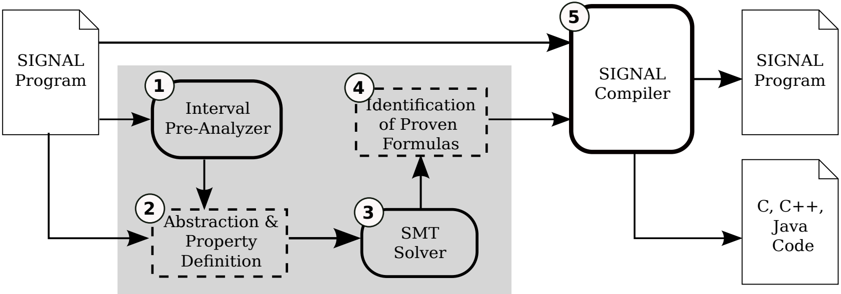
Description and analysis of a Bathtub (see paper)

- 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 of the paper



Identifying properties and their abstraction - WIP

Idea :

- do not use SMT-solver as blackbox anymore.
- use a SMT-simplifier !

Given Φ, an SMT solver/simplifier :

- construct a list of disjuncts subformulas.
- is able to detect for each subformula the list of false monoms.

► Get the list of false monoms for all subformulas (intersection)

On the example

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

- 14 models, only `balarm` is false in every model.

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)
- find more interesting properties in the SMT outputs.

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