An Alternative Polychronous Model and Synthesis Methodology for Model-Driven Embedded Software

Bijoy A. Jose   Sandeep K. Shukla

FERMAT Lab
Department of Electrical & Computer Engineering
Virginia Polytechnic and State University
shukla@vt.edu

January 19, 2010 / ASP-DAC.

Work Supported by AFOSR, AF Rome Labs, NSF

Outline

Outline

Outline

Outline

Outline

What is common among these?
Embedded Systems?

Software for Safety Critical systems

Requirements

- **Correctness – Functionality**
- **Timeliness – Real Time**
- **Reliability – Fault and Defect Tolerance**

Software for Safety Critical systems
Requirements

- **Correctness – Functionality**
- **Timeliness – Real Time**
- **Reliability – Fault and Defect Tolerance**

Software for Safety Critical systems

Requirements

- **Correctness – Functionality**
- **Timeliness – Real Time**
- **Reliability – Fault and Defect Tolerance**

C programming for Safety Critical systems

Requirements

- **Time reference**
- **Concurrency**
- **Rate of execution**

C programming for Safety Critical systems

Requirements

- Time reference
- **Concurrency**
- Rate of execution

C programming for Safety Critical systems

Requirements

- Time reference
- Concurrency
- **Rate of execution**

Model-Driven Code Generation

Specification Driven Correct-by-Construction Software Synthesis

- **State Machine based formalism – Statecharts (Stateflow -SIMULINK, LabVIEW)**
- **Synchronous Programming – ESTEREL, LUSTRE**
- **Polychronous formalism – SIGNAL**

Model-Driven Code Generation

Specification Driven Correct-by-Construction Software Synthesis

- State Machine based formalism – Statecharts (Stateflow -SIMULINK, LabVIEW)
- **Synchronous Programming – ESTEREL, LUSTRE**
- Polychronous formalism – SIGNAL

Model-Driven Code Generation

Specification Driven Correct-by-Construction Software Synthesis

- State Machine based formalism – Statecharts (Stateflow -SIMULINK, LabVIEW)
- Synchronous Programming – ESTEREL, LUSTRE
- **Polychronous formalism – SIGNAL**

Model-Driven Code Generation

Why Polychronous formalism ?

- **Processing speed decides rate of execution OR Environment decides rate of execution**
- **External trigger for computation OR Internal event triggered computation**

MRICDF formalism

Model-Driven Code Generation

Why Polychronous formalism ?

- **Processing speed decides rate of execution OR Environment decides rate of execution**
- **External trigger for computation OR Internal event triggered computation**

MRICDF formalism

Outline

MRICDF

Multi-Rate Instantaneous Channel Connected Data Flow actor network model

- **Visual representation of polychronous formalism**
- **Hierarchical representation of actor network**
- **Generate sequential C code from polychronous specification**

Outline

Signals and Events

Definition (Event)

An occurrence of a value or computational of a value is called an event

Definition (Signal)

A signal is a totally ordered sequence of events

Definition (Set of all Events)

$\mathcal{E}$ **– set of all events**

Definition (Event set of a signal)

$E(a)$ **– set of events on signal** a

Signals and Events

Definition (Event)

An occurrence of a value or computational of a value is called an event

Definition (Signal)

A signal is a totally ordered sequence of events

Definition (Set of all Events)

$\mathcal{E}$ **– set of all events**

Definition (Event set of a signal)

$E(a)$ **– set of events on signal** a

Signals and Events

Definition (Event)

An occurrence of a value or computational of a value is called an event

Definition (Signal)

A signal is a totally ordered sequence of events

Definition (Set of all Events)

$\mathcal{E}$ **– set of all events**

Definition (Event set of a signal)

$E(a)$ **– set of events on signal** a

Signals and Events

Definition (Event)

An occurrence of a value or computational of a value is called an event

Definition (Signal)

A signal is a totally ordered sequence of events

Definition (Set of all Events)

$\mathcal{E}$ **– set of all events**

Definition (Event set of a signal)

$E(a)$ **– set of events on signal** a

Sequencing events in a Signal

Definition (Event Sequence)

$$\sigma(a) : \aleph \longrightarrow \varepsilon$$

$\sigma(a)(i)$ – i^{th} **event of signal** a

Events are Partially Ordered

Definition (Partial Order on Events)

$\preceq \subseteq \varepsilon \times \varepsilon$ **is a** **partial order** **on** ε

$e \preceq f \Rightarrow f$ is either a prerequisite for e or synchronizable with e

Definition (Synchronizable events)

$$e \sim f$$

e **is a prerequisite for or synchronous with** f
and f **is a prerequisite for or synchronous with** e

$$\Rightarrow$$

e **and** f **are synchronous**

$$\sim \subseteq \varepsilon \times \varepsilon$$

an equivalence relation

Events are Partially Ordered

Definition (Partial Order on Events)

$\preceq \subseteq \varepsilon \times \varepsilon$ **is a partial order on** ε

$e \preceq f \Rightarrow f$ **is either a prerequisite for** e **or synchronizable with** e

Definition (Synchronizable events)

$e \sim f$

e **is a prerequisite for or synchronous with** f
and f **is a prerequisite for or synchronous with** e

$\Rightarrow$

e **and** f **are synchronous**

$\sim \subseteq \varepsilon \times \varepsilon$

an equivalence relation

Events are Partially Ordered

Definition (Partial Order on Events)

$\preceq \subseteq \varepsilon \times \varepsilon$ **is a partial order on** ε

$e \preceq f \Rightarrow f$ **is either a prerequisite for** e **or synchronizable with** e

Definition (Synchronizable events)

$e \sim f$

e **is a prerequisite for or synchronous with** f
and f **is a prerequisite for or synchronous with** e

$\Rightarrow$

e **and** f **are synchronous**

$\sim \subseteq \varepsilon \times \varepsilon$

an equivalence relation

Data Dependence

Definition (Data Dependence Relation)

$$\rightharpoondown \subseteq \varepsilon \times \varepsilon$$
$$\forall e, f \in \varepsilon \; e \rightharpoondown f \; \textbf{implies} \; e \preceq f$$

Instants without Timing

Definition (Instants)

$$\Upsilon = \varepsilon / \sim$$

$T \in \Upsilon$ – an **instant**

Definition (preorder among instants)

$$S, T \in \Upsilon$$

$S \prec T$ iff for all events $e \in S$, and $f \in T$, $e \prec f$

Instants without Timing

Definition (Instants)

$$\Upsilon = \varepsilon / \sim$$

$T \in \Upsilon$ **– an instant**

Definition (preorder among instants)

$S, T \in \Upsilon$

$S \prec T$ **iff for all events** $e \in S$**, and** $f \in T$**,** $e \prec f$

Definition (Signal Behavior)

$\beta(a)$ **– An infinite sequence (totally ordered) of events**

Definition (Signal Epoch)

$I(a) \subseteq \Upsilon$ **– set of instants at which** a **has an event**

$I(a)$ **– epoch of signal** a

Definition (Synchronous Signals)

$I(a), I(b) \subseteq \Upsilon$ **–** a **and** b **synchronous iff** $I(a) = I(b)$

Definition (Signal Behavior)

$\beta(a)$ **– An infinite sequence (totally ordered) of events**

Definition (Signal Epoch)

$I(a) \subseteq \Upsilon$ **– set of instants at which** a **has an event**

$I(a)$ **– epoch of signal** a

Definition (Synchronous Signals)

$I(a), I(b) \subseteq \Upsilon$ **–** a **and** b **synchronous iff** $I(a) = I(b)$

Definition (Signal Behavior)

$\beta(a)$ – **An infinite sequence (totally ordered) of events**

Definition (Signal Epoch)

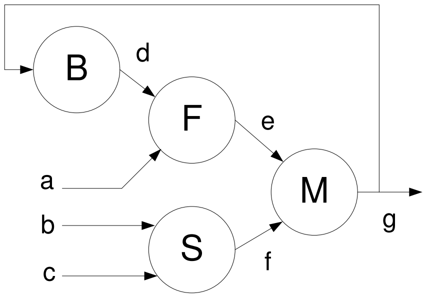
$I(a) \subseteq \Upsilon$ – **set of instants at which** a **has an event**
$I(a)$ – **epoch of signal** a

Definition (Synchronous Signals)

$I(a), I(b) \subseteq \Upsilon$ – a **and** b **synchronous iff** $I(a) = I(b)$

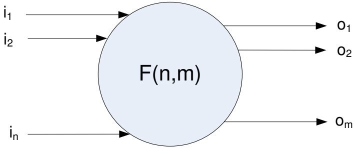
Outline

Multi-Rate Instantaneous Channel Connected Data Flow Actors



Actor network1
(input integer a,b; boolean c;
output integer g) =
(| d = Buffer(g)
| e = Function(a,d)
| f = Sampler(b,c)
| g = Merge(e,f)
|)

Primitive Actors: Function Actor



$$\forall I \in \aleph$$

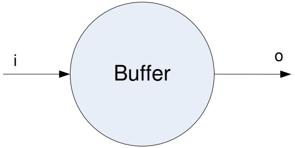
for an instant $S \in \Upsilon$

$$\forall j = \{1...n\}\ \sigma(i_j)(I) \in S$$

$$\forall k = \{1...m\}\ \sigma(o_k)(I) \in S$$

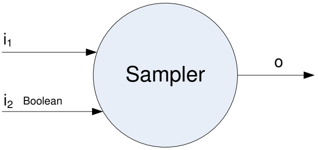
$$\langle \sigma(o_1)(I), \sigma(o_2)(I), ..., \sigma(o_m)(I) \rangle = F(\sigma(i_1)(I), \sigma(i_2)(I), ..., \sigma(i_n)(I)$$

Primitive Actors: Buffer Actor



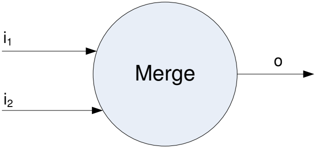
$$\forall I \in \aleph$$
$$\sigma(o)(I) \sim \sigma(i)(I)$$
$$val(\sigma(o)(I)) = val(\sigma(i)(I-1))$$
$\sigma(o)(1)$ **is a default value**

Primitive Actors: Sampler Actor



$\forall l \in \aleph$, $\exists j \in \aleph$ **and** $\exists T \in \Upsilon$
such that $\sigma(i_1)(l), \sigma(i_2)(j) \in T$
and
if $val(\sigma(i_2)(j)) = true$
$\exists k \in \aleph$ **such that** $\sigma(o)(k) \in T$ **with** $val(\sigma(o)(k)) = val(\sigma(i_1)(l))$
If $val(\sigma(i_2)(j)) = false$ **then** $\nexists k \in \aleph$ **such that** $\sigma(o)(k) \sim \sigma(i_1)(l)$
if $\sigma(i_1)(i) \in T$ **but** $\nexists j \sigma(i_2)(j) \in T$ **then** $\nexists k \in \aleph$ **such that** $\sigma(o)(k) \sim \sigma(i_1)(l)$

Primitive Actors: Priority Merge Actor



If for $k \in \aleph$, $\nexists j \in \aleph$ **such that** $\sigma(i_1)(k), \sigma(o)(j) \in S$ **for any** $S \in \Upsilon$ **then** $\exists l \in \aleph$ **such that** $\sigma(o)(l) \sim \sigma(i_1)(k)$ **and**

$$val(\sigma(o)(k)) = val(\sigma(i_1)(l))$$

$\exists j \in \aleph$, $\sigma(i_2)(j) \in S$ **for some** $S \in \Upsilon$, **and** $\exists i \in \aleph$ **such that** $\sigma(x)(i) \in S$ **then** $\exists l \in \aleph$ **such that** $\sigma(o)(l) \sim \sigma(i_2)(j)$ **where**

$$val(\sigma(o)(l)) = val(\sigma(i_2)(j))$$

Instantaneous Channels

If x and y connected by instantaneous channels then

$$\forall I \in \aleph$$

$$\sigma(x)(I), \sigma(y)(I) \in S \textbf{ for some } S \in \Upsilon$$

$$val(\sigma(x)(I)) = val(\sigma(y)(I))$$

- Function Actors ($\langle y_1, y_2, ..y_m\rangle := F(x_1, x_2, ..., x_n)$)
  - $I(x_1) = I(x_2) = ... = I(x_n) = I(y_1) = I(y_2) = ... = I(y_m)$
- Buffer Actor ($y := Buffer(x, 0)$)
  - $I(y) = I(x)$
- Sampling Actor ($y := Sampler(x, c)$)
  - $I(y) = I(x) \cap I([c])$
- Priority Merge Actor ($z := Merge(x, y)$)
  - $I(z) = I(x) \cup I(y)$

- Function Actors ($\langle y_1, y_2, ..y_m \rangle := F(x_1, x_2, ..., x_n)$)
  - $I(x_1) = I(x_2) = ... = I(x_n) = I(y_1) = I(y_2) = ... = I(y_m)$
- Buffer Actor ($y := Buffer(x, 0)$)
  - $I(y) = I(x)$
- Sampling Actor ($y := Sampler(x, c)$)
  - $I(y) = I(x) \cap I([c])$
- Priority Merge Actor ($z := Merge(x, y)$)
  - $I(z) = I(x) \cup I(y)$

- Function Actors ($\langle y_1, y_2, ..y_m\rangle := F(x_1, x_2, ..., x_n)$)
  - $I(x_1) = I(x_2) = ... = I(x_n) = I(y_1) = I(y_2) = ... = I(y_m)$
- Buffer Actor ($y := Buffer(x, 0)$)
  - $I(y) = I(x)$
- Sampling Actor ($y := Sampler(x, c)$)
  - $I(y) = I(x) \cap I([c])$
- Priority Merge Actor ($z := Merge(x, y)$)
  - $I(z) = I(x) \cup I(y)$

- Function Actors ($\langle y_1, y_2, ..y_m\rangle := F(x_1, x_2, ..., x_n)$)
  - $I(x_1) = I(x_2) = ... = I(x_n) = I(y_1) = I(y_2) = ... = I(y_m)$
- Buffer Actor ($y := Buffer(x, 0)$)
  - $I(y) = I(x)$
- Sampling Actor ($y := Sampler(x, c)$)
  - $I(y) = I(x) \cap I([c])$
- Priority Merge Actor ($z := Merge(x, y)$)
  - $I(z) = I(x) \cup I(y)$

User Specified Epoch Constraints – A Coordination Language

- User can specify various synchronization requirements by adding extra epoch Constraints
  - Signals a and b must always be synchronized – $I(a) = I(b)$
  - Events on signal a is always synchronized with events on signal b or on signal c – $I(a) = I(b) \cup I(c)$
  - Whenever a condition event (e.g $(x < 0)$, $c = true$) happens, events a and b must be synchronized – $I(a) = I(b) \cap I([cond])$

User Specified Epoch Constraints – A Coordination Language

- User can specify various synchronization requirements by adding extra epoch Constraints
  - Signals a and b must always be synchronized – $I(a) = I(b)$
  - Events on signal a is always synchronized with events on signal b or on signal c – $I(a) = I(b) \cup I(c)$
  - Whenever a condition event (e.g $(x < 0)$, $c = true$) happens, events a and b must be synchronized – $I(a) = I(b) \cap I([cond])$

User Specified Epoch Constraints – A Coordination Language

- User can specify various synchronization requirements by adding extra epoch Constraints
  - Signals *a* and *b* must always be synchronized – $I(a) = I(b)$
  - Events on signal *a* is always synchronized with events on signal *b* or on signal *c* – $I(a) = I(b) \cup I(c)$
  - Whenever a condition event (e.g $(x < 0)$, $c = true$) happens, events *a* and *b* must be synchronized – $I(a) = I(b) \cap I([cond])$

User Specified Epoch Constraints – A Coordination Language

- User can specify various synchronization requirements by adding extra epoch Constraints
  - Signals *a* and *b* must always be synchronized – $I(a) = I(b)$
  - Events on signal *a* is always synchronized with events on signal *b* or on signal *c* – $I(a) = I(b) \cup I(c)$
  - Whenever a condition event (e.g $(x < 0)$, $c = true$) happens, events *a* and *b* must be synchronized – $I(a) = I(b) \cap I([cond])$

Outline

Basic Ideas of Sequential Implementation

- So far events and instants are partially ordered – no global timing
- Partially ordered instants resulted from concurrency
- All events in one instant should be mapped to one round of computing
  - All data dependence must be respected within a round
- In sequential implementation computing rounds should progress linearly
  - Linear rounds $\rightarrow$: constructing total order out of partial order
  - Must not be imposed externally – must emerge from within

Basic Ideas of Sequential Implementation

- So far events and instants are partially ordered – no global timing
- Partially ordered instants resulted from concurrency
- All events in one instant should be mapped to one round of computing
  - All data dependence must be respected within a round
- In sequential implementation computing rounds should progress linearly
  - Linear rounds $\rightarrow$: constructing total order out of partial order
  - Must not be imposed externally – must emerge from within

Basic Ideas of Sequential Implementation

- So far events and instants are partially ordered – no global timing
- Partially ordered instants resulted from concurrency
- All events in one instant should be mapped to one round of computing
  - All data dependence must be respected within a round
- In sequential implementation computing rounds should progress linearly
  - Linear rounds $\rightarrow$: constructing total order out of partial order
  - Must not be imposed externally – must emerge from within

Basic Ideas of Sequential Implementation

- So far events and instants are partially ordered – no global timing
- Partially ordered instants resulted from concurrency
- All events in one instant should be mapped to one round of computing
  - All data dependence must be respected within a round
- In sequential implementation computing rounds should progress linearly
  - Linear rounds $\rightarrow$: constructing total order out of partial order
  - Must not be imposed externally – must emerge from within

Basic Ideas of Sequential Implementation

- So far events and instants are partially ordered – no global timing
- Partially ordered instants resulted from concurrency
- All events in one instant should be mapped to one round of computing
  - All data dependence must be respected within a round
- In sequential implementation computing rounds should progress linearly
  - Linear rounds $\rightarrow$: constructing total order out of partial order
  - Must not be imposed externally – must emerge from within

Basic Ideas of Sequential Implementation

- So far events and instants are partially ordered – no global timing
- Partially ordered instants resulted from concurrency
- All events in one instant should be mapped to one round of computing
    - All data dependence must be respected within a round
- In sequential implementation computing rounds should progress linearly
    - Linear rounds $\longrightarrow$: constructing total order out of partial order
    - Must not be imposed externally – must emerge from within

Sequential Implementability of a Specification

- Even though instants are partially ordered
  - Are the epoch constraints (implicit and explicit) implying a total order
  - If instants are in a total order, the rounds go linearly
  - We have our global totally ordered time

Sequential Implementability of a Specification

- Even though instants are partially ordered
  - Are the epoch constraints (implicit and explicit) implying a total order
  - If instants are in a total order, the rounds go linearly
  - We have our global totally ordered time

Sequential Implementability of a Specification

- Even though instants are partially ordered
  - Are the epoch constraints (implicit and explicit) implying a total order
  - If instants are in a total order, the rounds go linearly
  - We have our global totally ordered time

Sequential Implementability of a Specification

- Even though instants are partially ordered
  - Are the epoch constraints (implicit and explicit) implying a total order
  - If instants are in a total order, the rounds go linearly
  - We have our global totally ordered time

Observation

If a signal a's instant set $I(a) = \Upsilon$ then Υ must be totally ordered.

Proof.

Every event of a has a corresponding instant in Υ, and for every instant there is an event in a. Since $E(a)$ is totally ordered by $\prec$, so Υ is also totally ordered □

Observation

If a signal a's instant set $I(a) = \Upsilon$ then Υ must be totally ordered.

Proof.

Every event of a has a corresponding instant in Υ, and for every instant there is an event in a. Since $E(a)$ is totally ordered by $\prec$, so Υ is also totally ordered □

a that linearizes Υ is called the Master Trigger

If an MRICDF model is sequentially implementable, it should have a Master Trigger. $\leftarrow$ A necessary condition.

There should be no instantaneous cyclic dependency (deadlock) $\leftarrow$ Another necessary condition.

a that linearizes Υ is called the Master Trigger

If an MRICDF model is sequentially implementable, it should have a Master Trigger. $\leftarrow$ A necessary condition.

There should be no instantaneous cyclic dependency (deadlock) $\leftarrow$ Another necessary condition.

a that linearizes Υ is called the Master Trigger

If an MRICDF model is sequentially implementable, it should have a Master Trigger. $\leftarrow$ A necessary condition.

There should be no instantaneous cyclic dependency (deadlock) $\leftarrow$ Another necessary condition.

Deadlock

Definition (Possible Deadlock)

If the instants are linearizable, then within each instant, the $\rightharpoondown$ relation may have a cycle. This may imply deadlock.

Example

$$
\begin{aligned}
X &:= Sampler(F(Y), U) \\
Z &:= Sampler(X, C) \\
U &:= Sampler(Z, P)
\end{aligned}
$$

Outline

Strategy

- **How do we deal with infinite sets ($\varepsilon, \Upsilon, E(a), I(a)$)**
- **We need to deal with finite objects during analysis**
- **Plain Old Set Theory provides a way**
  - **To prove $S = T$ for two sets S and T**
  - **prove $S \subseteq T$ and $T \subseteq S$**
  - **To prove $S \subseteq T$**
  - **Take an arbitrary $s \in S$ and prove $s \in T$**

Strategy

- **How do we deal with infinite sets ($\varepsilon, \Upsilon, E(a), I(a)$)**
- **We need to deal with finite objects during analysis**
- **Plain Old Set Theory provides a way**
  - **To prove $S = T$ for two sets S and T**
  - **prove $S \subseteq T$ and $T \subseteq S$**
  - **To prove $S \subseteq T$**
  - **Take an arbitrary $s \in S$ and prove $s \in T$**

Strategy

- **How do we deal with infinite sets ($\varepsilon, \Upsilon, E(a), I(a)$)**
- **We need to deal with finite objects during analysis**
- **Plain Old Set Theory provides a way**
  - **To prove $S = T$ for two sets S and T**
  - **prove $S \subseteq T$ and $T \subseteq S$**
  - **To prove $S \subseteq T$**
  - **Take an arbitrary $s \in S$ and prove $s \in T$**

Strategy

- **How do we deal with infinite sets ($\varepsilon, \Upsilon, E(a), I(a)$)**
- **We need to deal with finite objects during analysis**
- **Plain Old Set Theory provides a way**
  - **To prove $S = T$ for two sets S and T**
  - **prove $S \subseteq T$ and $T \subseteq S$**
  - **To prove $S \subseteq T$**
  - **Take an arbitrary $s \in S$ and prove $s \in T$**

Strategy

- **How do we deal with infinite sets ($\varepsilon, \Upsilon, E(a), I(a)$)**
- **We need to deal with finite objects during analysis**
- **Plain Old Set Theory provides a way**
  - **To prove $S = T$ for two sets S and T**
  - **prove $S \subseteq T$ and $T \subseteq S$**
  - **To prove $S \subseteq T$**
  - **Take an arbitrary $s \in S$ and prove $s \in T$**

Strategy

- **How do we deal with infinite sets ($\varepsilon, \Upsilon, E(a), I(a)$)**
- **We need to deal with finite objects during analysis**
- **Plain Old Set Theory provides a way**
  - **To prove $S = T$ for two sets S and T**
  - **prove $S \subseteq T$ and $T \subseteq S$**
  - **To prove $S \subseteq T$**
  - **Take an arbitrary $s \in S$ and prove $s \in T$**

Strategy

- **How do we deal with infinite sets ($\varepsilon, \Upsilon, E(a), I(a)$)**
- **We need to deal with finite objects during analysis**
- **Plain Old Set Theory provides a way**
  - **To prove $S = T$ for two sets S and T**
  - **prove $S \subseteq T$ and $T \subseteq S$**
  - **To prove $S \subseteq T$**
  - **Take an arbitrary $s \in S$ and prove $s \in T$**

Apply Strategy

- **Surely by definition $I(a) \subseteq \Upsilon$**
- **We need to check if $\Upsilon \subseteq I(a)$**
- **Consider an arbitrary instant $T \in \Upsilon$ and show $T \in I(a)$**
  - **For each signal a, attach a Boolean b_a to mean $T \in I(a)$**
  - **Recall $\Upsilon = \cup_s I(s)$**
  - **$T \in \Upsilon$ implies $T \in I(s)$ for some s ; Thus $b_s = true$**
  - **if for signal a for all s we show $b_s \rightarrow b_a$**
  - **Then for all $T \in \Upsilon$ also $T \in I(a) \Rightarrow \Upsilon \subseteq I(a)$**

Apply Strategy

- **Surely by definition $I(a) \subseteq \Upsilon$**
- **We need to check if $\Upsilon \subseteq I(a)$**
- **Consider an arbitrary instant $T \in \Upsilon$ and show $T \in I(a)$**
  - **For each signal a, attach a Boolean b_a to mean $T \in I(a)$**
  - **Recall $\Upsilon = \cup_s I(s)$**
  - **$T \in \Upsilon$ implies $T \in I(s)$ for some s ; Thus $b_s = true$**
  - **if for signal a for all s we show $b_s \rightarrow b_a$**
  - **Then for all $T \in \Upsilon$ also $T \in I(a) \Rightarrow \Upsilon \subseteq I(a)$**

Apply Strategy

- **Surely by definition** $I(a) \subseteq \Upsilon$
- **We need to check if** $\Upsilon \subseteq I(a)$
- **Consider an arbitrary instant** $T \in \Upsilon$ **and show** $T \in I(a)$
  - **For each signal** a**, attach a Boolean** b_a **to mean** $T \in I(a)$
  - **Recall** $\Upsilon = \cup_s I(s)$
  - $T \in \Upsilon$ **implies** $T \in I(s)$ **for some** s **; Thus** $b_s = true$
  - **if for signal** a **for all** s **we show** $b_s \rightarrow b_a$
  - **Then for all** $T \in \Upsilon$ **also** $T \in I(a) \Rightarrow \Upsilon \subseteq I(a)$

Apply Strategy

- **Surely by definition** $I(a) \subseteq \Upsilon$
- **We need to check if** $\Upsilon \subseteq I(a)$
- **Consider an arbitrary instant** $T \in \Upsilon$ **and show** $T \in I(a)$
  - **For each signal** a**, attach a Boolean** b_a **to mean** $T \in I(a)$
  - **Recall** $\Upsilon = \cup_s I(s)$
  - $T \in \Upsilon$ **implies** $T \in I(s)$ **for some** s **; Thus** $b_s = true$
  - **if for signal** a **for all** s **we show** $b_s \rightarrow b_a$
  - **Then for all** $T \in \Upsilon$ **also** $T \in I(a) \Rightarrow \Upsilon \subseteq I(a)$

Apply Strategy

- **Surely by definition** $I(a) \subseteq \Upsilon$
- **We need to check if** $\Upsilon \subseteq I(a)$
- **Consider an arbitrary instant** $T \in \Upsilon$ **and show** $T \in I(a)$
  - **For each signal** a**, attach a Boolean** b_a **to mean** $T \in I(a)$
  - **Recall** $\Upsilon = \cup_s I(s)$
  - $T \in \Upsilon$ **implies** $T \in I(s)$ **for some** s **; Thus** $b_s = true$
  - **if for signal** a **for all** s **we show** $b_s \rightarrow b_a$
  - **Then for all** $T \in \Upsilon$ **also** $T \in I(a) \Rightarrow \Upsilon \subseteq I(a)$

Apply Strategy

- **Surely by definition** $I(a) \subseteq \Upsilon$
- **We need to check if** $\Upsilon \subseteq I(a)$
- **Consider an arbitrary instant** $T \in \Upsilon$ **and show** $T \in I(a)$
  - **For each signal** a**, attach a Boolean** b_a **to mean** $T \in I(a)$
  - **Recall** $\Upsilon = \cup_s I(s)$
  - $T \in \Upsilon$ **implies** $T \in I(s)$ **for some** s **; Thus** $b_s = true$
  - **if for signal** a **for all** s **we show** $b_s \rightarrow b_a$
  - **Then for all** $T \in \Upsilon$ **also** $T \in I(a) \Rightarrow \Upsilon \subseteq I(a)$

Apply Strategy

- **Surely by definition** $I(a) \subseteq \Upsilon$
- **We need to check if** $\Upsilon \subseteq I(a)$
- **Consider an arbitrary instant** $T \in \Upsilon$ **and show** $T \in I(a)$
  - **For each signal** a**, attach a Boolean** b_a **to mean** $T \in I(a)$
  - **Recall** $\Upsilon = \cup_s I(s)$
  - $T \in \Upsilon$ **implies** $T \in I(s)$ **for some** s **; Thus** $b_s = true$
  - **if for signal** a **for all** s **we show** $b_s \rightarrow b_a$
  - **Then for all** $T \in \Upsilon$ **also** $T \in I(a) \Rightarrow \Upsilon \subseteq I(a)$

Apply Strategy

- **Surely by definition** $I(a) \subseteq \Upsilon$
- **We need to check if** $\Upsilon \subseteq I(a)$
- **Consider an arbitrary instant** $T \in \Upsilon$ **and show** $T \in I(a)$
  - **For each signal** a**, attach a Boolean** b_a **to mean** $T \in I(a)$
  - **Recall** $\Upsilon = \cup_s I(s)$
  - $T \in \Upsilon$ **implies** $T \in I(s)$ **for some** s **; Thus** $b_s = true$
  - **if for signal** a **for all** s **we show** $b_s \rightarrow b_a$
  - **Then for all** $T \in \Upsilon$ **also** $T \in I(a) \Rightarrow \Upsilon \subseteq I(a)$

Epoch Equations to Boolean Equations

- **if** $I(x) = I(b)$ **then we write** $b_x = b_y$
- **if** $I(x) = I(y) \cup I(z)$ **we write** $b_x = b_y \vee b_z$
- **If** $I(x) = I(y) \cap I(z)$ **we write** $b_x = b_y \wedge b_z$
- **For a Boolean signal** c**, we write** $b_c = b_{[c]} \vee b_{[\neg c]}$ **and** $b_{[c]} \wedge b_{[\neg c]} = \mathit{false}$

F **denotes the system of Boolean Equations. If defines a Boolean Relation** R

Epoch Equations to Boolean Equations

- **if** $I(x) = I(b)$ **then we write** $b_x = b_y$
- **if** $I(x) = I(y) \cup I(z)$ **we write** $b_x = b_y \vee b_z$
- **If** $I(x) = I(y) \cap I(z)$ **we write** $b_x = b_y \wedge b_z$
- **For a Boolean signal** c**, we write** $b_c = b_{[c]} \vee b_{[\neg c]}$ **and** $b_{[c]} \wedge b_{[\neg c]} = false$

F **denotes the system of Boolean Equations. If defines a Boolean Relation** R

Epoch Equations to Boolean Equations

- **if** $I(x) = I(b)$ **then we write** $b_x = b_y$
- **if** $I(x) = I(y) \cup I(z)$ **we write** $b_x = b_y \vee b_z$
- **If** $I(x) = I(y) \cap I(z)$ **we write** $b_x = b_y \wedge b_z$
- **For a Boolean signal** c**, we write** $b_c = b_{[c]} \vee b_{[\neg c]}$ **and** $b_{[c]} \wedge b_{[\neg c]} = false$

F **denotes the system of Boolean Equations. If defines a Boolean Relation** R

Epoch Equations to Boolean Equations

- **if** $I(x) = I(b)$ **then we write** $b_x = b_y$
- **if** $I(x) = I(y) \cup I(z)$ **we write** $b_x = b_y \vee b_z$
- **If** $I(x) = I(y) \cap I(z)$ **we write** $b_x = b_y \wedge b_z$
- **For a Boolean signal** c**, we write** $b_c = b_{[c]} \vee b_{[\neg c]}$ **and** $b_{[c]} \wedge b_{[\neg c]} = false$

F **denotes the system of Boolean Equations. If defines a Boolean Relation** R

Inferring Master Trigger

Observation

If for all s $b_s \rightarrow b_x$ *then when* $b_x = false$*, for all s,* $b_s = false$ *for* F *to be satisfied.*

Observation

$F \cup \{\neg x, \vee_{s \neq b_x} b_s\}$ *is UNSAT.*

Prime Implicates of Boolean Theories

Definition (Boolean Theory)

A set of Boolean equations F defines a theory Σ
Σ – the set of all satisfying assignments of F

Definition (Prime Implicate)

A disjunctive clause C with $\Sigma \models C$ is an implicate of Σ
If $\nexists C'$ such that $\Sigma \models C' \models C$ then C is a prime implicate of Σ

Definition (Unitary Implicate)

If a prime implicate C is a single literal then C is a unitary implicate

Observation

A unitary implicate is always prime

Observation

If x is a variable and a prime implicate of F
For all variables y

$$y \longrightarrow x$$

Observation

V ***– set of Boolean variables in*** F

$F' = F \wedge (\vee_{b_v \in V} b_v)$

If x ***corresponds to master trigger, then*** $F' \models b_x$

b_x ***is unitary prime implicate of*** F'

Observation

For every $b_s \in V$ ***we have*** $b_s \rightarrow b_x$

Finding the next Trigger

Definition

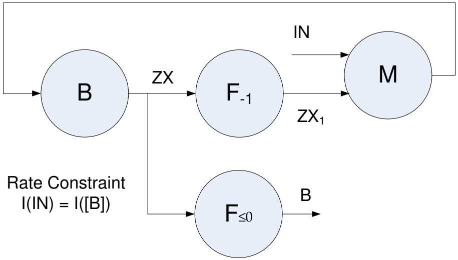
If x is found to be a trigger, and Y is the set of signals which must trigger next, then Y is the next trigger set.

Observation

$F = (b_x \wedge F_{b_x=1}) \vee (\bar{b_x} \wedge F_{b_x=0})$ ***by Shannon decomposition.***
Consider $F' = F_{b_x=1} \wedge \bigvee_{b_v \in V \setminus \{b_x\}} b_v$
If C ***is a prime implicate, then*** C ***indicates set of next triggers***

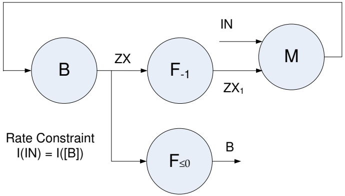
Outline

Stop Watch



X, ZX and $ZX1$ are internal registers. X gets the input count, ZX holds the value of X from previous computation, $ZX1$ holds the result of $ZX - 1$. X is allowed to input new value only when previous count reached 0.

Example



Example (Stop Watch)

$$X := Merge(IN, ZX1)$$
$$ZX1 = f_{-1}(ZX)$$
$$ZX = Buffer(X, 0)$$
$$B := f_{\leq 0}(ZX)$$
$$I(IN) = I([B])$$

Example (Equations)

$$b_X = b_{IN} \vee b_{ZX1}$$
$$b_{ZX1} = b_{ZX}$$
$$b_{ZX} = b_X$$
$$b_B = b_{ZX}$$
$$b_{IN} = b_{[B]}$$
$$b_B = b_{[B]} \vee b_{[\neg B]}$$

Example (cont..)

Example (Add $B_I N \vee$ ..)

$b_x = b_{IN} \vee b_{ZX1}$
$b_{ZX1} = b_{ZX}$
$b_{ZX} = b_X$
$b_B = b_{ZX}$
$b_{IN} = b_{[B]}$
$b_B = b_{[B]} \vee b_{[\neg B]}$
$b_B \vee b_I N \vee b_X \vee b_{ZX} \vee b_{ZX1} \vee b_{[B]} \vee b_{[\neg B]}$

4 Unitary Implicates

$b_X, b_{ZX}, b_{ZX1}, b_B$

X is master trigger

$ZX, ZX1, B$ are synchronous with X

Example (cont...)

Example (Set $b_x = 1$)

$b_x = b_{IN} \vee b_{ZX1}$
$b_{ZX1} = b_{ZX}$
$b_{ZX} = b_X$
$b_B = b_{ZX}$
$b_{IN} = b_{[B]}$
$b_B = b_{[B]} \vee b_{[\neg B]}$
$b_B \vee b_I N \vee b_X \vee b_{ZX} \vee b_{ZX1} \vee$
$b_{[B]} \vee b_{[\neg B]}$

Example

$b_{IN} = b_{[B]}$
$1 = b_{[B]} \vee b_{[\neg B]}$
Add $b_{IN} \vee b_{[B]} \vee b_{[\neg B]}$

Prime Implicate $b_{[B]} \vee b_{[\neg B]}$

Example(cont....)

Split Cases

Example (Set $b_{[B]} = 1$)

$b_{IN} = 1$
Add $b_{IN} \vee b_{[\neg B]}$

Prime Implicate b_{IN}

Other Case

Example (Set $b_{[\neg B]} = 1$)

$b_{IN} = 0$ Add $b_{IN} \vee b_{[B]}$

prime implicate $\neg b_{IN}$

Code Generation

```
int X = 0;
int ZX = 0;
int ZX1 := 0;
bool B = false;
int IN;

While(true){
    B:= (ZX <= 0) ? true: false;
    if (B)  X := read(IN);
    print(X);
    ZX = X;
    ZX1 = ZX -1;
    X = ZX1;
}
```

Code Generation

```
int X = 0;
int ZX = 0;
int ZX1 := 0;
bool B = false;
int IN;

While(true){
      B:= (ZX <= 0) ? true: false;
      if (B)  X := read(IN);
      print(X);
      ZX = X;
      ZX1 = ZX -1;
      X = ZX1;
}
```

X	ZX	ZX1
0	0	0
5	5	4
4	4	3
3	3	2
2	2	1
1	1	0
0	0	-1

Summary

- In this paper we only present sequential software synthesis
- We contend that for concurrent specification language assuming a priori a global time is not natural
- We contend that a priori global timeline in the specification misses out optimizations in the synthesized software
- Correct of Construction Software synthesis can be enabled by Polychrony
- Existing Polychronous frameworks are too complicated for engineers to use
- We provide an alternative model, alternative interpretation of semantics and alternative synthesis algorithms

Ongoing Work

- We have created a Software synthesis tool to generate C code - EmCodeSyn [FDL2009]
- We are developing an actor elimination technique to reduce complexity of generation prime implicates (E-SAT)
- We have modeled real life aviation modules such as Flight warning system, read blackboard service, etc.
- Other interests include Multi-threaded Code Synthesis, Real-Time Scheduling, etc.