Model-Driven Embedded Software Generation: A Generative Approach to Safety

Sandeep K. Shukla *

*FERMAT Lab
Virginia Polytechnic and State University
shukla@vt.edu
Work Supported by AFOSR, AF Rome Labs, NSF

IEEE Computer Society Distinguished Visitor Lecture Series
July/Aug 2010 / Canada
Outline

1. Introduction
   - Examples of Safety Critical System
   - Safety Critical System Design Practices

2. Correct-by-Construction Synthesis from Models
   - Alternatives?
Outline

1. Introduction
   - Examples of Safety Critical System
   - Safety Critical System Design Practices

2. Correct-by-Construction Synthesis from Models
   - Alternatives?
1. Introduction
   - Examples of Safety Critical System
   - Safety Critical System Design Practices

2. Correct-by-Construction Synthesis from Models
   - Alternatives?
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
Outline

1. Introduction
   - Examples of Safety Critical System
   - Safety Critical System Design Practices

2. Correct-by-Construction Synthesis from Models
   - Alternatives?
How do we design embedded software today?
A Simple Programming Model

```plaintext
code
loop
  read inputs/sensors;
  compute response;
  generate actuator outputs;
forever
```
Design Decisions

- How to read inputs?
- How often to read inputs?
- In What order to read the inputs?
- How to compute responses?
- How often to generate responses?
A Time Triggered Approach

```
loop
  await tick;
  read S_1; take-action;
  read S_2; take-action;
  read S_3; take-action;
forever
```
A Time Triggered Approach

```plaintext
loop
  await tick;
  read S_1; take-action;
  read S_2; take-action;
  read S_3; take-action;
forever
```

Time is globally linearized from outside
Problems?

- Processing speed decides the input rate!
  - Fine for interactive systems not for reactive systems
- It should be the other way around:
  - Characters coming through network interface card
  - Streaming video frames
  - Signals from pacemaker
- All sensors are treated identically
  - Some require urgent processing
- Fragile scheme
  - More sensors $\rightarrow$ more processing delay
A General Scheme

\[ Task_1 \| Task_2 \| \cdots \| Task_n \]

Tasks:
- Sequential Threads
- Concurrently Executed
- Communicate with each other
- Wait for specific time periods or events
- Can be scheduled and suspended
Real-Time O/S

- Manages tasks
- Facilitates task communications
- Provides Timer services
- Schedules tasks based on scheduling strategies
Issues?

- More complex design: reign in nondeterminism
- Writing and understanding concurrent tasks very difficult
- Race conditions, deadlocks, live-locks
What Else?

- Building predictable (deterministic) system challenging—nondeterminism
- Posteriori Verification – State space explosion
- Simulation based verification – Mars Rover?
Outline

1 Introduction
   - Examples of Safety Critical System
   - Safety Critical System Design Practices

2 Correct-by-Construction Synthesis from Models
   - Alternatives?
What’s the alternative?

Specification Driven Correct-by-Construction Software Synthesis
Few Alternative Modeling Paradigms

- **Domain Specific Languages**
  - SDF, cyclo-static dataflow, DSMLs

- **Formal Specification Languages**
  - Temporal Logics, TLA+
  - Action Oriented Specification Languages, UNITY
  - I/O automata, Interacting automata, S/R Language, State charts
  - Process algebras

- **Synchronous Languages**
  - Imperative Synchronous – Esterel, Quartz
  - Declarative Dataflow synchronous – Lustre, SCADE

- **Multi-rate Languages** – SIGNAL/Polychrony, MRICDF
Model of Time

- Is Time Globally Linear inside a Program?
- Do you want to impose Global Linear Time line on the execution of your program?
  - Do you lose anything if you do so?
  - Optimization opportunities perhaps?
  - Can we keep opportunities to exploit concurrency without losing determinism?
  - What if we want sequential program?
Let’s Consider Examples

\begin{tabular}{c|c|c|c|c|c|c|c|c}
  \hline
  a & T & \_ & \_ & T & \_ & \_ & T & T \\
  \hline
  b & \_ & \_ & T & \_ & \_ & \_ & T & T \\
  \hline
  c & T & \_ & T & T & T & \_ & \_ & T \\
  \hline
\end{tabular}
Introduction
Model Based Synthesis
Semantic Concepts
Synthesis from MRICDF
Summary

Alternatives?
Onto Multi-Rate
Main Topic of this Presentation

MEMORY

Buffer

Consumer

a

b

1

2

1

1

1

1

1

1

1

1

2

A New Polychrony Based Code Synthesis Approach
RAZOR

A New Polychrony Based Code Synthesis Approach
A New Polychrony Based Code Synthesis Approach
- Eliminate Global Time Line from the Specification Paradigm
- Define a Dataflow Language for Specification – MRICDF
- Abstract Semantics of MRICDF
- Sequential Implementability and Reconstruction of Global Time line
- Sequential Software Synthesis from MRICDF
### Signals and Events

<table>
<thead>
<tr>
<th>Definition (Event)</th>
<th>An occurrence of a value or computational of a value is called an <strong>event</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition (Signal)</td>
<td>A <strong>signal</strong> is a totally ordered sequence of events</td>
</tr>
<tr>
<td>Definition (Set of all Events)</td>
<td>$\mathcal{E}$ – set of all events</td>
</tr>
<tr>
<td>Definition (Event set of a signal)</td>
<td>$E(a)$ – set of events on signal $a$</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Definition (Event)</th>
<th>An occurrence of a value or computational of a value is called an <strong>event</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition (Signal)</td>
<td>A <strong>signal</strong> is a totally ordered sequence of events</td>
</tr>
<tr>
<td>Definition (Set of all Events)</td>
<td>$\mathcal{E}$ – set of all events</td>
</tr>
<tr>
<td>Definition (Event set of a signal)</td>
<td>$E(a)$ – set of events on signal $a$</td>
</tr>
</tbody>
</table>
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)

\[ \sigma(a) : \mathbb{N} \rightarrow \varepsilon \]

\[ \sigma(a)(i) – i^{th} \text{ event of signal } a \]
Events are Partially Ordered

Definition (Partial Order on Events)

\[
\leq \subseteq \varepsilon \times \varepsilon \text{ is a partial order on } \varepsilon
\]

\[
e \leq f \Rightarrow f \text{ is either a prerequisite for } e \text{ or synchronizable with } e
\]

Definition (Synchronizable events)

\[
e \sim f
\]

\[
e \text{ is a prerequisite for or synchronous with } f \text{ and } f \text{ is a prerequisite for or synchronous with } e
\]

\[
\Rightarrow
\]

\[
e \text{ and } f \text{ are synchronous}
\]
Events are Partially Ordered

Definition (Partial Order on Events)
\[ \leq \subseteq \varepsilon \times \varepsilon \text{ is a partial order on } \varepsilon \]
\[ e \leq f \Rightarrow f \text{ is either a prerequisite for } e \text{ or synchronizable with } e \]

Definition (Synchronizable events)
\[ e \sim f \]
\[ e \text{ is a prerequisite for or synchronous with } f \]
\[ \text{and } f \text{ is a prerequisite for or synchronous with } e \]
\[ \Rightarrow \]
\[ e \text{ and } f \text{ are synchronous} \]
Events are Partially Ordered

Definition (Partial Order on Events)
\[ \leq \subseteq \varepsilon \times \varepsilon \text{ is a partial order on } \varepsilon \]

\( e \leq f \Rightarrow f \text{ is either a prerequisite for } e \text{ or synchronizable with } e \)

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
Definition (Sychronicity is an Equivalence Relation)

\[ \sim \subseteq \epsilon \times \epsilon \]

an equivalence relation
Definition (Data Dependence Relation)

\[ \rightarrow \subseteq \mathcal{E} \times \mathcal{E} \]

\[ \forall e, f \in \mathcal{E}, e \rightarrow f \implies e \preceq f \]
Instants without Timing

Definition (Instants)
\[ \Upsilon = \varepsilon / \sim \]
\[ T \in \Upsilon \text{ – an instant} \]

Definition (preorder among instants)
\[ S, T \in \Upsilon \]
\[ S \prec T \text{ iff for all events } e \in S, \text{ and } f \in T, \; e \prec f \]
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)

$\beta(a) - \text{An infinite sequence (totally ordered) of events}$

Definition (Signal Epoch)

$I(a) \subseteq \Upsilon - \text{set of instants at which } a \text{ has an event}$

$I(a) - \text{epoch of signal } a$

Definition (Synchronous Signals)

$I(a), I(b) \subseteq \Upsilon - a \text{ and } b \text{ synchronous iff } I(a) = I(b)$
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 \[ I(a) = I(b) \]
Multi-Rate Instantaneous Channel Connected Data Flow Actors

Actor 1

Actor 2

Actor 3

Actor 4

Signals, Events, MRICDF Models Sequential Implementation: Issues
Primitive Actors: Function Actor

∀\_l \in \mathbb{N}

for an instant \_S \in \Upsilon

∀\_j = \{1...n\} \_\sigma(i_j)(\_l) \in \_S

∀\_k = \{1...m\} \_\sigma(o_k)(\_l) \in \_S

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

S. Shukla
A New Polychrony Based Code Synthesis Approach
Primitive Actors: Buffer Actor

∀l ∈ ℕ

σ(o)(l) ~ σ(i)(l)

val(σ(o)(l)) = val(σ(i)(l - 1))

σ(o)(1) is a default value
Primitive Actors: Sampler Actor

∀l ∈ ℕ, ∃j ∈ ℕ and ∃T ∈ ℳ such that σ(i_1)(l), σ(i_2)(j) ∈ T and

if \( \text{val}(σ(i_2)(j)) = true \)

∃k ∈ ℕ such that \( σ(o)(k) ∈ T \) with \( \text{val}(σ(o)(k)) = \text{val}(σ(i_1)(l)) \)

If \( \text{val}(σ(i_2)(j)) = false \) then \( ∄k ∈ ℕ \) such that \( σ(o)(k) ∼ σ(i_1)(l) \)

if \( σ(i_1)(i) ∈ T \) but \( ∄jσ(i_2)(j) ∈ T \) then \( ∄k ∈ ℕ \) such that

\( σ(o)(k) ∼ σ(i_1)(l) \)
If for $k \in \mathbb{N}$, $\not\exists j \in \mathbb{N}$ such that $\sigma(i_1)(k), \sigma(o)(j) \in S$ for any $S \in \Upsilon$ then $\exists l \in \mathbb{N}$ such that $\sigma(o)(l) \sim \sigma(i_1)(k)$ and $\text{val}(\sigma(o)(k)) = \text{val}(\sigma(i_1)(l))$.

$\exists j \in \mathbb{N}, \sigma(i_2)(j) \in S$ for some $S \in \Upsilon$, and $\exists i \in \mathbb{N}$ such that $\sigma(x)(i) \in S$ then $\exists l \in \mathbb{N}$ such that $\sigma(o)(l) \sim \sigma(i_2)(j)$ where $\text{val}(\sigma(o)(l)) = \text{val}(\sigma(i_2)(j))$.
If $x$ and $y$ connected by instantaneous channels then

$$\forall \ell \in \mathbb{N} \quad \sigma(x)(\ell), \sigma(y)(\ell) \in S \text{ for some } S \in \mathcal{Y}$$

$$\text{val}(\sigma(x)(\ell)) = \text{val}(\sigma(y)(\ell))$$
Function Actors ($\langle y_1, y_2, \ldots, y_m \rangle := F(x_1, x_2, \ldots, x_n)$)

- $I(x_1) = I(x_2) = \ldots = I(x_n) = I(y_1) = I(y_2) = \ldots = I(y_m)$

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 \((\langle y_1, y_2, \ldots, y_m \rangle := F(x_1, x_2, \ldots, x_n))\)
- \(I(x_1) = I(x_2) = \ldots = I(x_n) = I(y_1) = I(y_2) = \ldots = I(y_m)\)

Buffer Actor \((y := \text{Buffer}(x, 0))\)
- \(I(y) = I(x)\)

Sampling Actor \((y := \text{Sampler}(x, c))\)
- \(I(y) = I(x) \cap I([c])\)

Priority Merge Actor \((z := \text{Merge}(x, y))\)
- \(I(z) = I(x) \cup I(y)\)
Function Actors \( \langle y_1, y_2, \ldots, y_m \rangle := F(x_1, x_2, \ldots, x_n) \)
- \( I(x_1) = I(x_2) = \ldots = I(x_n) = I(y_1) = I(y_2) = \ldots = I(y_m) \)

Buffer Actor (\( y := \text{Buffer}(x, 0) \))
- \( I(y) = I(x) \)

Sampling Actor (\( y := \text{Sampler}(x, c) \))
- \( I(y) = I(x) \cap I([c]) \)

Priority Merge Actor (\( z := \text{Merge}(x, y) \))
- \( I(z) = I(x) \cup I(y) \)
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 – $l(a) = l(b)$
  - Events on signal $a$ is always synchronized with events on signal $b$ or on signal $c$ – $l(a) = l(b) \cup l(c)$
  - Whenever a condition event (e.g $(x < 0)$, $c = true$) happens, events $a$ and $b$ must be synchronized – $l(a) = l(b) \cap l([cond])$
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 – $I(a) = I(b) \cap I([cond])$
Basic Ideas of Sequential Implementation

- So far events and instants are partially ordered – no global timing
- Partly 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 → 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 →: 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 → : 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 → 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 → : constructing total order out of partial order
  - Must not be imposed externally – must emerge from within
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
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
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
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 \( \Upsilon \) must be totally ordered.*

Proof.

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

If a signal $a$’s instant set $I(a) = \succ$ then $\succ$ must be totally ordered.

Proof.

Every event of $a$ has a corresponding instant in $\succ$, and for every instant there is an event in $a$. Since $E(a)$ is totally ordered by $\prec$, so $\succ$ is also totally ordered.
a that linearizes $\gamma$ is called the Master Trigger

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

There should be no instantaneous cyclic dependency (deadlock) ← Another necessary condition.
a that linearizes $\gamma$ is called the **Master Trigger**

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

There should be no instantaneous cyclic dependency (**deadlock**) ← Another necessary condition.
a that linearizes $\Upsilon$ is called the **Master Trigger**.

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

There should be no instantaneous cyclic dependency (deadlock) ← Another necessary condition.
Deadlock

Definition (Possible Deadlock)
If the instants are linearizable, then within each instant, the \(\rightarrow\) relation may have a cycle. This may imply deadlock.

Example

\[
\begin{align*}
X & := \text{Sampler}(F(Y), U) \\
Z & := \text{Sampler}(X, C) \\
U & := \text{Sampler}(Z, P)
\end{align*}
\]
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 $s \in T$
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 \((\varepsilon, \gamma, E(a), I(a))\)?
- 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 \(s \in T\)
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 \((\varepsilon, \gamma, E(a), I(a))\)?
- 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 \((\varepsilon, \gamma, E(a), I(a))\)?
- 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 \(T\)
  - \(S \subseteq T\) 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 ($\varepsilon, \cap, E(a), I(a)$)?
- 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 $T$
  - proove $S \subseteq T$ and $T \subseteq S$
  - To prove $S \subseteq T$
  - Take an arbitrary $s \in S$ and prove $s \in T$
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 $\Upsilon = \bigcup_s I(s)$
- $T \in \Upsilon$ implies $T \in I(s)$ for some $s$; Thus $b_s = \text{true}$
- 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 $\Upsilon = \bigcup_s I(s)$
  - $T \in \Upsilon$ 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 $\Upsilon = \bigcup_s I(s)$
  - $T \in \Upsilon$ implies $T \in I(s)$ for some $s$; Thus $b_s = \text{true}$
  - if for signal $a$ for all $s$ we show $b_s \rightarrow b_a$
  - Then for all $T \in \Upsilon$ also $T \in I(a) \Rightarrow \Upsilon \subseteq I(a)$
Surely by definition $l(a) \subseteq \Upsilon$

We need to check if $\Upsilon \subseteq l(a)$

Consider an arbitrary instant $T \in \Upsilon$ and show $T \in l(a)$

- For each signal $a$, attach a Boolean $b_a$ to mean $T \in l(a)$
  - Recall $\Upsilon = \bigcup_s l(s)$
  - $T \in \Upsilon$ implies $T \in l(s)$ 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 $T \in l(a) \Rightarrow \Upsilon \subseteq l(a)$
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 \( \Upsilon = \bigcup_s I(s) \)

\( T \in \Upsilon \) 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 \( T \in I(a) \Rightarrow \Upsilon \subseteq I(a) \)
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 $b_s = \text{true}$

if for signal $a$ for all $s$ we show $b_s \rightarrow b_a$

Then for all $T \in \Upsilon$ also $T \in I(a) \Rightarrow \Upsilon \subseteq I(a)$
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 \( T \in I(a) \implies \Upsilon \subseteq I(a) \)
Surely by definition \( l(a) \subseteq \Upsilon \)

We need to check if \( \Upsilon \subseteq l(a) \)

Consider an arbitrary instant \( T \in \Upsilon \) and show \( T \in l(a) \)

- For each signal \( a \), attach a Boolean \( b_a \) to mean \( T \in l(a) \)
- Recall \( \Upsilon = \bigcup_s I(s) \)
- \( T \in \Upsilon \) implies \( T \in I(s) \) for some \( s \); Thus \( b_s = \text{true} \)
- if for signal \( a \) for all \( s \) we show \( b_s \rightarrow b_a \)
- Then for all \( T \in \Upsilon \) also \( T \in l(a) \Rightarrow \Upsilon \subseteq l(a) \)
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 $b_x = b_y \lor b_z$
- If $I(x) = I(y) \cap I(z)$ we write $b_x = b_y \land b_z$
- For a Boolean signal $c$, we write $b_c = b[c] \lor b[\neg c]$ and $b[c] \land b[\neg c] = false$

$F$ 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** \( b_x = b_y \lor b_z \)

- **If** \( I(x) = I(y) \cap I(z) \) **we write** \( b_x = b_y \land b_z \)

- **For a Boolean signal** \( c \), **we write** \( b_c = b_{[c]} \lor b_{[\neg c]} \) **and** \( b_{[c]} \land b_{[\neg c]} = false \)

\( F \) 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 \( b_x = b_y \lor b_z \)
- If \( I(x) = I(y) \cap I(z) \) we write \( b_x = b_y \land b_z \)
- For a Boolean signal \( c \), we write \( b_c = b_[c] \lor b_[\neg c] \) and \( b_[c] \land b_[\neg c] = false \)

\( F \) 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 $b_x = b_y \lor b_z$
- If $I(x) = I(y) \cap I(z)$ we write $b_x = b_y \land b_z$
- For a Boolean signal $c$, we write $b_c = b[c] \lor b[\neg c]$ and $b[c] \land b[\neg c] = \text{false}$

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

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

**Observation**

$F \cup \{-x, \lor_{s \neq b_x} b_s\}$ is UNSAT.
Definition (Boolean Theory)

A set of Boolean equations $F$ defines a theory $\Sigma$.
$\Sigma$ – the set of all satisfying assignments of $F$.

Definition (Prime Implicate)

A disjunctive clause $C$ with $\Sigma \models C$ is an implicate of $\Sigma$.
If $\not\exists C'$ such that $\Sigma \models C' \models C$ then $C$ is a prime implicate of $\Sigma$. 

S. Shukla  
A New Polychrony Based Code Synthesis Approach
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 $y$*

\[ y \rightarrow x \]
Observation

\( V \) – set of Boolean variables in \( F \)

\[ F' = F \land \left( \lor_{b_v \in V} b_v \right) \]

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
\[
F = (b_x \land F_{b_x=1}) \lor (\bar{b}_x \land F_{b_x=0}) \text{ by Shannon decomposition.}
\]
Consider \( F' = F_{b_x=1} \lor \bigvee_{b_v \in V \setminus \{b_x\}} b_v \)
If \( C \) is a prime implicate, then \( C \) indicates set of next triggers.
Boolean Equations

\[ B, F-1, M, F \leq 0, IN, ZX, 1 \]

\[ X, ZX, ZX1 \] 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)**

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

**Example (Equations)**

\[ b_x = b_{IN} \lor b_{ZX1} \]
\[ b_{ZX1} = b_{ZX} \]
\[ b_{ZX} = b_X \]
\[ b_B = b_{ZX} \]
\[ b_{IN} = b_{[B]} \]
\[ b_B = b_{[B]} \lor b_{[\neg B]} \]

---

S. Shukla

A New Polychrony Based Code Synthesis Approach
4 Unitary Implicates

\( b_X, b_{ZX}, b_{ZX1}, b_B \)

\( X \) is master trigger

\( ZX, ZX1, B \) are synchronous with \( X \)
Example (Set $b_x = 1$)

- $b_x = b_{IN} \lor b_{ZX1}$
- $b_{ZX1} = b_{ZX}$
- $b_{ZX} = b_X$
- $b_B = b_{ZX}$
- $b_{IN} = b_{[B]}$
- $b_B = b_{[B]} \lor b_{[\neg B]}$
- $b_B \lor b_{IN} \lor b_X \lor b_{ZX} \lor b_{ZX1} \lor b_{[B]} \lor b_{[\neg B]}$

Example

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

**Prime Implicate** $b_{[B]} \lor b_{[\neg B]}$
Example (cont....)

Split Cases

Example (Set $b_B = 1$)

\[ b_{IN} = 1 \]
\[ \text{Add } b_{IN} \lor b_{\neg B} \]

Prime Implicate $b_{IN}$

Other Case

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

\[ b_{IN} = 0 \]
\[ \text{Add } b_{IN} \lor b_B \]

prime implicate $\neg b_{IN}$
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;
}
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;
}
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

```c
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;
}
```
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;
}

S. Shukla
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;
}
### Boolean Equations

<table>
<thead>
<tr>
<th>X</th>
<th>ZX</th>
<th>ZX1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>
### Example Run

<table>
<thead>
<tr>
<th>X</th>
<th>ZX</th>
<th>ZX1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>X</td>
<td>ZX</td>
<td>ZX1</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>-----</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>
### Example Run

<table>
<thead>
<tr>
<th>X</th>
<th>ZX</th>
<th>ZX1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>
### Example Run

<table>
<thead>
<tr>
<th>X</th>
<th>ZX</th>
<th>ZX1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>
### Example Run

<table>
<thead>
<tr>
<th>X</th>
<th>ZX</th>
<th>ZX1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>
### Example Run

<table>
<thead>
<tr>
<th>X</th>
<th>ZX</th>
<th>ZX1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>
## Boolean Equations

<table>
<thead>
<tr>
<th>$X$</th>
<th>ZX</th>
<th>ZX1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>
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.

In this talk we only present sequential software synthesis.
Ongoing Work

- Optimization of synthesized code
- Multi-threaded Code Synthesis
- Real-Time Scheduling
- Hardware synthesis with Low-Power Techniques
- Adding fault-tolerance during synthesis