A Temporal Language for SystemC

Deian Tabakov, Moshe Y. Vardi, Gila Kamhi, Eli Singerman

Rice University
Houston, TX

November 20, 2008
SystemC

- System-level modeling language: C++ based, OO used for abstraction, modularity, and compositionality
- Rich set of data types: C++ plus hardware
- Rich set of libraries for modeling at different levels: signals, FIFOs, TLM (transaction-level modeling)
- Processes; SC_METHODs and SC_THREADs
- Simulation kernel – event driven
  - Processes run until suspension
  - Processes notify events (immediate, delta, timed)
  - Notified events wake suspended processes
  - Kernel manages scheduling of processes
<table>
<thead>
<tr>
<th>Producer</th>
<th>signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Kernel: make all processes active</strong></td>
</tr>
</tbody>
</table>

- **SystemC**
- **Producer** sends an *int* to **Consumer**.
- **Kernel** statement explains the process activity.
**Producer** | **signal** | **Consumer**
---|---|---
*Kernel: make all processes active*

*skip*
Producer \[\text{int}\] Consumer

<table>
<thead>
<tr>
<th>Producer</th>
<th>signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>–</td>
<td></td>
</tr>
</tbody>
</table>

*Kernel: make all processes active*
**SystemC**

![Diagram](image)

<table>
<thead>
<tr>
<th>Producer</th>
<th>signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>–</td>
<td>wait(value_changed)</td>
</tr>
</tbody>
</table>

*Kernel: make all processes active*
<table>
<thead>
<tr>
<th>Producer</th>
<th>signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Kernel: make all processes active</strong></td>
</tr>
<tr>
<td>skip</td>
<td>-</td>
<td>wait(value_changed)</td>
</tr>
<tr>
<td></td>
<td><strong>Kernel: make Producer active</strong></td>
<td></td>
</tr>
<tr>
<td>Producer</td>
<td>signal</td>
<td>Consumer</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>Kernel: make all processes active</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>skip</td>
<td>-</td>
<td>wait(value_changed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Kernel: make Producer active**
SystemC

Producer \(\text{signal} \) Consumer

- \text{Producer: make all processes active}
- \text{Kernel: make Producer active}
- \text{write(1)}
- \text{wait(value_changed)}
- \text{start} \(\Delta\)
- \text{end} \(\Delta\)
The diagram shows a SystemC producer-consumer relationship with an integer signal. The producer initiates an event by signaling to the consumer. The consumer responds by waiting for a value change, then starting a process, writing a value of 1, and notifying the producer of the change. The kernel makes all processes active.

<table>
<thead>
<tr>
<th>Producer</th>
<th>signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>skip</td>
<td>–</td>
<td>wait(value_changed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(1)</td>
<td>notify(value_changed)</td>
<td></td>
</tr>
</tbody>
</table>

**Kernel:** make all processes active

**Kernel:** make Producer active
<table>
<thead>
<tr>
<th>Producer</th>
<th>signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>–</td>
<td>wait(value_changed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>start Δ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end Δ</td>
</tr>
<tr>
<td>write(1)</td>
<td>notify(value_changed)</td>
<td>sleeping</td>
</tr>
</tbody>
</table>

*Kernel: make all processes active*

*Kernel: make Producer active*
### SystemC

**Producer** \( \rightarrow \text{int} \rightarrow \text{Consumer} \)

<table>
<thead>
<tr>
<th>Producer</th>
<th>signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>skip</strong></td>
<td>–</td>
<td><strong>wait</strong>(value_changed)</td>
</tr>
<tr>
<td><strong>write(1)</strong></td>
<td>notify(value_changed)</td>
<td><strong>sleeping</strong></td>
</tr>
</tbody>
</table>

*Kernel: make all processes active*

*Kernel: make Producer active*

*Kernel: update value of signal*
<table>
<thead>
<tr>
<th>Producer</th>
<th>signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>–</td>
<td>wait(value_changed)</td>
</tr>
<tr>
<td>write(1)</td>
<td>notify(value_changed)</td>
<td>sleeping</td>
</tr>
</tbody>
</table>

*Kernel: make all processes active*

*Kernel: make Producer active*

*Kernel: update value of signal*

*Kernel: Make Producer and Consumer active*
Deian Tabakov, Moshe Y. Vardi, Gila Kamhi, Eli Singerman

A Temporal Language for SystemC

<table>
<thead>
<tr>
<th>Producer</th>
<th>signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel: make all processes active</td>
<td></td>
<td></td>
</tr>
<tr>
<td>skip</td>
<td>–</td>
<td>wait(value_changed)</td>
</tr>
<tr>
<td>kernel: make Producer active</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(1)</td>
<td>notify(value_changed)</td>
<td>sleeping</td>
</tr>
<tr>
<td>kernel: update value of signal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kernel: Make Producer and Consumer active</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The diagram shows a flowchart with two processes, Producer and Consumer, connected by an 'int' signal. The Producer process sends an integer to the Consumer process. The table summarizes the actions taken in the Producer and Consumer processes, including making processes active, waiting for signals, and notifying other processes of changes.
SystemC

![SystemC Diagram]

<table>
<thead>
<tr>
<th>Producer</th>
<th>signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>–</td>
<td>wait(value_changed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>start Δ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end Δ</td>
</tr>
<tr>
<td>write(1)</td>
<td>notify(value_changed)</td>
<td>sleeping</td>
</tr>
<tr>
<td></td>
<td></td>
<td>start Δ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>update</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end Δ</td>
</tr>
</tbody>
</table>

Kernel: make all processes active

Kernel: make Producer active

Kernel: update value of signal

Kernel: Make Producer and Consumer active
### SystemC

![Diagram of SystemC producer and consumer](image)

<table>
<thead>
<tr>
<th>Producer</th>
<th>signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kernel:</strong> make all processes active</td>
<td></td>
<td></td>
</tr>
<tr>
<td>skip</td>
<td>–</td>
<td><strong>wait</strong>(value_changed)</td>
</tr>
<tr>
<td><strong>Kernel:</strong> make Producer active</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>write</strong>(1)</td>
<td>notify(value_changed)</td>
<td>sleeping</td>
</tr>
<tr>
<td><strong>Kernel:</strong> update value of signal</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Kernel:</strong> Make Producer and Consumer active</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>write</strong>(2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Deian Tabakov, Moshe Y. Vardi, Gila Kamhi, Eli Singerman  
A Temporal Language for SystemC
## SystemC

![Diagram showing the interaction between Producer and Consumer]

### Table: Producer-Consumer Interaction

<table>
<thead>
<tr>
<th>Producer</th>
<th>signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Kernel:** make all processes active

<table>
<thead>
<tr>
<th>skip</th>
<th>–</th>
<th>wait(value_changed)</th>
<th>start Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>end Δ</td>
</tr>
</tbody>
</table>

**Kernel:** make Producer active

<table>
<thead>
<tr>
<th>write(1)</th>
<th>notify(value_changed)</th>
<th>sleeping</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Kernel:** update value of signal

**Kernel:** Make Producer and Consumer active

<table>
<thead>
<tr>
<th>write(2)</th>
<th>1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### SystemC

**Diagram:**

```
Producer -> int -> Consumer
```

<table>
<thead>
<tr>
<th>Producer signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kernel: make all processes active</strong></td>
<td></td>
</tr>
<tr>
<td>skip</td>
<td>wait(value_changed)</td>
</tr>
<tr>
<td></td>
<td>start Δ</td>
</tr>
<tr>
<td></td>
<td>Kernel: make Producer active</td>
</tr>
<tr>
<td></td>
<td>wait(value_changed)</td>
</tr>
<tr>
<td></td>
<td>start Δ</td>
</tr>
<tr>
<td><strong>Kernel: update value of signal</strong></td>
<td></td>
</tr>
<tr>
<td>write(1)</td>
<td>notify(value_changed)</td>
</tr>
<tr>
<td></td>
<td>Kernel: Make Producer and Consumer active</td>
</tr>
<tr>
<td></td>
<td>start Δ</td>
</tr>
<tr>
<td></td>
<td>end Δ</td>
</tr>
<tr>
<td></td>
<td>update</td>
</tr>
<tr>
<td></td>
<td>end Δ</td>
</tr>
<tr>
<td>write(2)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Kernel: Make Producer and Consumer active</td>
</tr>
<tr>
<td></td>
<td>start Δ</td>
</tr>
<tr>
<td></td>
<td>end Δ</td>
</tr>
<tr>
<td>read(1)</td>
<td></td>
</tr>
</tbody>
</table>
SystemC

![Diagram](image)

<table>
<thead>
<tr>
<th>Producer</th>
<th>signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kernel: make all processes active</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>skip</td>
<td>–</td>
<td>wait(value_changed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kernel: make Producer active</td>
</tr>
<tr>
<td>write(1)</td>
<td>notify(value_changed)</td>
<td>sleeping</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kernel: update value of signal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kernel: Make Producer and Consumer active</td>
</tr>
<tr>
<td>write(2)</td>
<td>1</td>
<td>read(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kernel: Make Producer and Consumer active</td>
</tr>
</tbody>
</table>
**SystemC**

![SystemC Diagram]

<table>
<thead>
<tr>
<th>Producer</th>
<th>signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Kernel: make all processes active**

<table>
<thead>
<tr>
<th>skip</th>
<th>–</th>
<th>wait(value_changed)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>start ( \Delta )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end ( \Delta )</td>
</tr>
</tbody>
</table>

**Kernel: make Producer active**

<table>
<thead>
<tr>
<th>write(1)</th>
<th>notify(value_changed)</th>
<th>sleeping</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>start ( \Delta )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>update</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end ( \Delta )</td>
</tr>
</tbody>
</table>

**Kernel: update value of signal**

**Kernel: Make Producer and Consumer active**

<table>
<thead>
<tr>
<th>write(2)</th>
<th>1</th>
<th>read(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>start ( \Delta )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end ( \Delta )</td>
</tr>
</tbody>
</table>
### SystemC Producer to Consumer

![Producer to Consumer Diagram](image)

<table>
<thead>
<tr>
<th>Producer</th>
<th>signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel: make all processes active</td>
<td></td>
<td></td>
</tr>
<tr>
<td>skip</td>
<td>–</td>
<td>wait(value_changed)</td>
</tr>
<tr>
<td>Kernel: make Producer active</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(1)</td>
<td>notify(value_changed)</td>
<td>sleeping</td>
</tr>
<tr>
<td>Kernel: update value of signal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel: Make Producer and Consumer active</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(2)</td>
<td>1</td>
<td>read(1)</td>
</tr>
<tr>
<td>Kernel: Make Producer and Consumer active</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...
SystemC

Producer \[\text{int}\] Consumer

Producer \[\text{signal}\] Consumer
SystemC

Producer

int

Consumer

write(1);

Producer

signal

Consumer
SystemC

Deian Tabakov, Moshe Y. Vardi, Gila Kamhi, Eli Singerman

A Temporal Language for SystemC

4/22
SystemC

Deian Tabakov, Moshe Y. Vardi, Gila Kamhi, Eli Singerman

A Temporal Language for SystemC
SystemC

Producer  \( \rightarrow \)  int  \( \rightarrow \)  Consumer

<table>
<thead>
<tr>
<th>Producer</th>
<th>signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>write(1); write(2);</td>
<td>notify(value_changed)</td>
<td>wait()</td>
</tr>
</tbody>
</table>
Producer

int

Consumer

write(1); write(2);

notify(value_changed)

wait()

Kernel: update value of signal
```plaintext

<table>
<thead>
<tr>
<th>Producer</th>
<th>signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>write(1);</td>
<td>notify(value_changed)</td>
<td>wait()</td>
</tr>
<tr>
<td>write(2);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Kernel: update value of signal*

*Kernel: Make Producer and Consumer active*
```
Producer

write(1); write(2); notify(value_changed)

Consumer

Kernel: update value of signal
Kernel: Make Producer and Consumer active

skip
SystemC

Producer \[\text{int} \rightarrow\] Consumer

<table>
<thead>
<tr>
<th>Producer</th>
<th>signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>write(1); write(2);</td>
<td>notify(value_changed)</td>
<td>wait()</td>
</tr>
<tr>
<td><strong>Kernel:</strong> update value of signal</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Kernel:</strong> Make Producer and Consumer active</td>
<td></td>
<td></td>
</tr>
<tr>
<td>skip</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
SystemC

Producer → int → Consumer

<table>
<thead>
<tr>
<th>Producer</th>
<th>signal</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>write(1); write(2);</td>
<td>notify(value_changed)</td>
<td>wait()</td>
</tr>
<tr>
<td>Kernel: update value of signal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel: Make Producer and Consumer active</td>
<td></td>
<td></td>
</tr>
<tr>
<td>skip</td>
<td>2</td>
<td>read(2)</td>
</tr>
</tbody>
</table>
Desideratum: Express properties at sub-$\Delta$-cycle resolution
SystemC Events

Three types of `sc_event`

- **Immediate events** have an immediate effect
  - Can cause deadlocks
- **Delta events**
  - Accumulated while processes are running
  - Have an effect only after all immediate events
- **Timed events**
  - Accumulated while processes are running
  - Have an effect only after all delta events
Three types of `sc_event`

- Immediate events have an immediate effect
  - Can cause deadlocks
- Delta events
  - Accumulated while processes are running
  - Have an effect only after all immediate events
- Timed events
  - Accumulated while processes are running
  - Have an effect only after all delta events

**Key observation:** No canonical notion of a cycle
• A signal is written to at most once

• The value of variable “balance” is always equal to “deposits” - “withdrawals”

• “Request” → within[3] “grant”
A signal is written to at most once within . . .

- . . . execution of an individual process
- . . . a complete delta cycle
- . . . between two clock ticks

The value of variable “balance” is always equal to “deposits” - “withdrawals”

“Request” $\rightarrow$ within[3] “grant”
A signal is written to at most once within . . .
- . . . execution of an individual process
- . . . a complete delta cycle
- . . . between two clock ticks

The value of variable “balance” is always equal to “deposits” - “withdrawals” . . .
- . . . in all stable states (no process running)
- . . . at beginning of each delta cycle
- . . . at each clock tick

“Request” → within[3] “grant”
A signal is written to at most once within ... execution of an individual process, a complete delta cycle, between two clock ticks.

The value of variable “balance” is always equal to “deposits” - “withdrawals” ...
  ... in all stable states (no process running)
  ... at beginning of each delta cycle
  ... at each clock tick

“Request” → within[3] “grant”
  Within 3 “what”? 
Recognize of SystemC’s ability to bridge different levels of abstraction
- Specify clockless and clocked modules working together
- Systematic way to refine properties as design is refined

Recognize SystemC’s unique simulation semantics
- Expose notification of events
- Allow different levels temporal resolution

Give precise definition of a trace
- No existing language addresses this issue
Augment existing languages (PSL/SVA), not develop a new one

- Define a precise notion of a trace of execution for SystemC models
- Identify important Boolean properties relevant to execution or specification of SystemC

Plug-in our framework in existing specification languages

- Richer set of Boolean properties
- Much more flexible temporal resolution: by leveraging the ability of temporal languages to use Boolean expressions as clock expressions
Why deal with the kernel?

- Many important properties at sub-$\Delta$ cycle resolution
- Adapt specifications to level of abstraction

Example: Invariance properties, say, $\text{ALWAYS } x > 10$

- Must hold at \textit{all} times
- Must hold when processes suspend
- Must hold at delta-cycle boundary
- Must hold at clock-cycle boundary

This is possible \textit{only} if we require the kernel to expose information about its internal state.
Complications

- Many implementations
- 15K lines of code (in reference implementation)
- What is the right abstraction?

Our solution

- Follow the LRM
- Abstract kernel’s implementations, but expose semantics
- Enable coarser abstractions via clock expression
- Expose event notifications
Complications

- Many implementations
- 15K lines of code (in reference implementation)
- What is the right abstraction?

Our solution

- Follow the LRM
- Abstract kernel’s implementations, but expose semantics
- Enable coarser abstractions via clock expression
- Expose event notifications

**Bottom line:** expose kernel state and event notifications
Dealing with user code

```
while ( true ) {
    wait(in.value_changed_event);
    int x = in.read();
    int y = f(x); // some one-way function
    float z = 10/y;
    ...
}
```
Dealing with user code

**Code (Consumer.h)**

```c
while ( true ) {
    wait(in.value_changed_event);
    int x = in.read();
    int y = f(x);  // some one-way function
    float z = 10/y;
    ...
}
```

**Desideratum:** Statement-level assertions
Dealing with user code

Our approach

- Each statement defines a new state
- Expose protected and private variables (white box)
- Expose properties of function calls (arguments and return value)
- Expose properties of SystemC primitives (e.g., number of elements in a `sc_fifo`)
A SystemC trace is a sequence of states corresponding to execution of model.

Exposne alternation of control:
- kernel
- user code
- libraries

“large-step semantics” vs “small-step semantics”
- \[ y = (x++) + (x--) \]
Balance = Deposits - Withdrawals
Refining specifications

- Balance = Deposits - Withdrawals
  - At end of all \( \Delta \) cycles
Refining specifications

- Balance = Deposits - Withdrawals
  - At end of all $\Delta$ cycles
  - When no process is running
Refining specifications

- Balance = Deposits - Withdrawals
  - At end of all $\Delta$ cycles
  - When no process is running
  - At end of a particular process
Balance = Deposits - Withdrawals

- At end of all Δ cycles
- When no process is running
- At end of a particular process
- When a particular function returns
Refining specifications

- \[ \text{Balance} = \text{Deposits} - \text{Withdrawals} \]
  - At end of all \( \Delta \) cycles
  - When no process is running
  - At end of a particular process
  - When a particular function returns

- Process A must execute
Refining specifications

- Balance = Deposits - Withdrawals
  - At end of all \( \Delta \) cycles
  - When no process is running
  - At end of a particular process
  - When a particular function returns

- Process A must execute
  - Every delta cycle
Balance = Deposits - Withdrawals

- At end of all $\Delta$ cycles
- When no process is running
- At end of a particular process
- When a particular function returns

Process A must execute

- Every delta cycle
- Every clock cycle
Refining specifications

- **Balance** = Deposits - Withdrawals
  - At end of all Δ cycles
  - When no process is running
  - At end of a particular process
  - When a particular function returns

- **Process A must execute**
  - Every delta cycle
  - Every clock cycle
  - Every 10 clock cycles
Refining specifications

- Balance = Deposits - Withdrawals
  - At end of all $\Delta$ cycles
  - When no process is running
  - At end of a particular process
  - When a particular function returns

- Process A must execute
  - Every delta cycle
  - Every clock cycle
  - Every 10 clock cycles
  - Only once
HW/SW co-design: Treating SystemC as software

- Pre- and post- conditions
- Properties about the actual parameters of function calls
- Properties about the return values of function calls
- Library state only via APIs

Relevant prior work

- SLIC and Blast allow the specification of C interfaces via specifying properties related to function calls
- Blast allows access to syntax of executing statements
HW/SW co-design: Treating SystemC as software

- Pre- and post- conditions
- Properties about the actual parameters of function calls
- Properties about the return values of function calls
- Library state only via APIs

Relevant prior work

- SLIC and Blast allow the specification of C interfaces via specifying properties related to function calls
- Blast allows access to syntax of executing statements

**Key Observation:** Expose the syntax
Summary

- Precise definition of an execution trace
- A family of expressions that enrich the Boolean layer of any specification language
- Mechanism for sampling underlying trace at different levels of abstraction without changing language
- SystemC as software

Discussion

- Framework applicable to formal and dynamic verification
- Our approach requires very small modifications of SystemC kernel
- Current focus: translate specifications into SystemC monitors and instrument user code
1: $PC \leftarrow \text{all primitive channels}$  
2: $P \leftarrow \text{all processes}$  
3: $R \leftarrow P$ /* Set of runnable processes */  
4: $D \leftarrow \emptyset$ /* Set of pending delta notifications */  
5: $U \leftarrow \emptyset$ /* Set of update requests */  
6: $T \leftarrow \emptyset$ /* Set of pending timed notifications */  
7: for all $chan \in PC$ do  
8: run $chan$.update()  
9: for all $p \in R$ do  
10: if $p$ is initializable then  
11: run $p$  
12: for all $d \in D$ do  
13: $D \leftarrow D \setminus d$  
14: for all $p \in P$ do  
15: if $n$ triggers $p$ then  
16: $R \leftarrow R \cup p$
17: repeat
18:   while $R \neq \emptyset$ do /* New delta cycle begins */
19:       for all $r \in R$ do
20:           $R \leftarrow R \setminus r$
21:           run $r$ until it invokes wait() or returns
22:       for all $chan \in U$ do /* Update phase */
23:           run $chan.update()$
24:       for all $d \in D$ do /* Delta notification phase */
25:           $D \leftarrow D \setminus d$
26:       for all $p \in P$ do
27:           if $d$ triggers $p$ then
28:               $R \leftarrow R \cup p$ /* $p$ is now runnable */
29: /* End of delta cycle */
30: if $T \neq \emptyset$ then
31: Advance clock to earliest timed delay $t$.
32: $T \leftarrow T \setminus t$
33: for all $p \in P$ do
34: if $t$ triggers $p$ then
35: $R \leftarrow R \cup p$ /* $p$ is now runnable */
36: until end of simulation
Our approach: keep track of current phase

Figure: Captured Kernel States
Kernel States in Moy’s Abstraction

Figure: Kernel states proposed by Moy et al.