A Temporal Language for SystemC

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

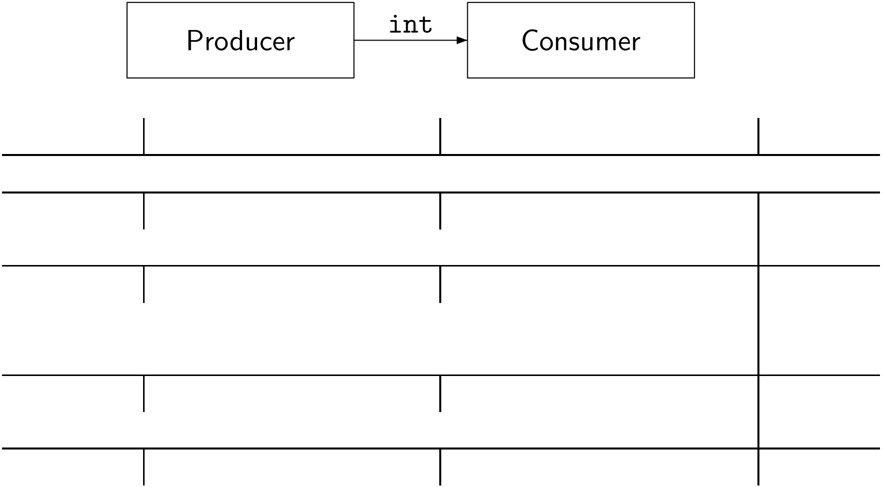
Rice University
Houston, TX

November 20, 2008

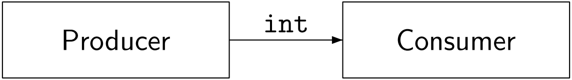
SystemC

- 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 deferent 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

SystemC



SystemC



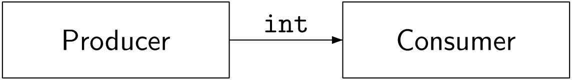
Producer

int

Consumer

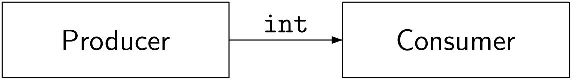
Producer	signal	Consumer	

SystemC



Producer	signal	Consumer	
Kernel: make all processes active			

SystemC



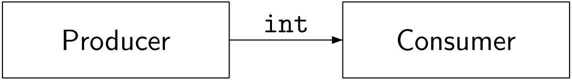
Producer

int

Consumer

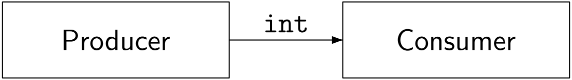
Producer	signal	Consumer
Kernel: make all processes active		
skip		

SystemC



Producer	signal	Consumer
Kernel: make all processes active		
skip	–	

SystemC

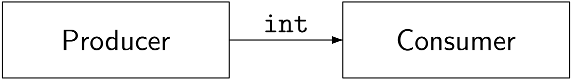

Producer

int

Consumer

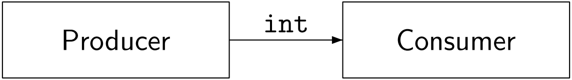
Producer	signal	Consumer	
Kernel: make all processes active			
skip	–	`wait(value_changed)`	

SystemC



Producer	signal	Consumer	
Kernel: make all processes active			
skip	–	`wait(value_changed)`	
Kernel: make Producer active			

SystemC

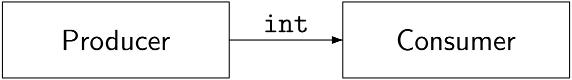

Producer

int

Consumer

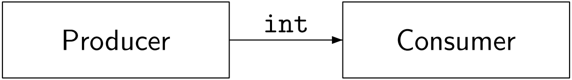
Producer	signal	Consumer	
Kernel: make all processes active			
skip	–	`wait(value_changed)`	start Δ
Kernel: make Producer active			end Δ

SystemC



Producer	signal	Consumer	
Kernel: make all processes active			
skip	–	`wait(value_changed)`	start Δ
Kernel: make Producer active			end Δ
write(1)			

SystemC

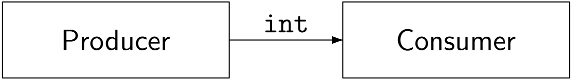

Producer

`int`

Consumer

Producer	signal	Consumer	
Kernel: make all processes active			
skip	–	`wait(value_changed)`	start Δ
Kernel: make Producer active			end Δ
write(1)	notify(value_changed)		

SystemC



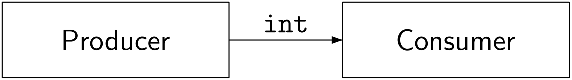
Producer

`int`

Consumer

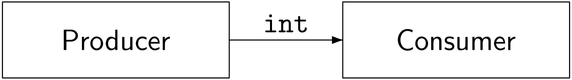
Producer	signal	Consumer	
Kernel: make all processes active			
skip	–	`wait(value_changed)`	start Δ
Kernel: make Producer active			end Δ
write(1)	notify(value_changed)	sleeping	

SystemC



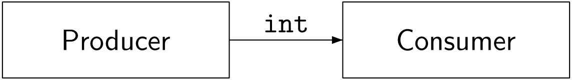
Producer	signal	Consumer	
Kernel: make all processes active			
skip	–	`wait(value_changed)`	start Δ
Kernel: make Producer active			end Δ
write(1)	notify(value_changed)	sleeping	
Kernel: update value of signal			

SystemC



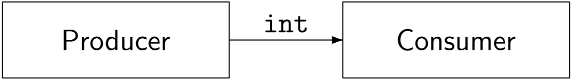
Producer	signal	Consumer	
Kernel: make all processes active			
skip	–	`wait(value_changed)`	start Δ
Kernel: make Producer active			end Δ
write(1)	notify(value_changed)	sleeping	
Kernel: update value of signal			
Kernel: Make Producer and Consumer active			

SystemC



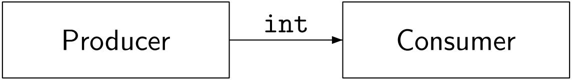
Producer	signal	Consumer	
Kernel: make all processes active			
skip	–	`wait(value_changed)`	start Δ
Kernel: make Producer active			end Δ
write(1)	notify(value_changed)	sleeping	start Δ
Kernel: update value of signal			
Kernel: Make Producer and Consumer active			end Δ

SystemC



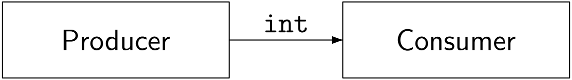
Producer	signal	Consumer	
Kernel: make all processes active			
skip	–	`wait(value_changed)`	start Δ
Kernel: make Producer active			end Δ
write(1)	notify(value_changed)	sleeping	start Δ
Kernel: update value of signal			update
Kernel: Make Producer and Consumer active			end Δ

SystemC



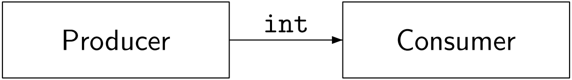
Producer	signal	Consumer	
Kernel: make all processes active			
skip	–	`wait(value_changed)`	start Δ
Kernel: make Producer active			end Δ
write(1)	notify(value_changed)	sleeping	start Δ
Kernel: update value of signal			update
Kernel: Make Producer and Consumer active			end Δ
write(2)			

SystemC



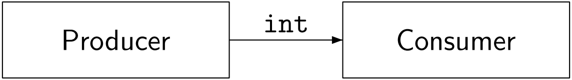
Producer	signal	Consumer	
Kernel: make all processes active			
skip	–	`wait(value_changed)`	start Δ
Kernel: make Producer active			end Δ
write(1)	notify(value_changed)	sleeping	start Δ
Kernel: update value of signal			update
Kernel: Make Producer and Consumer active			end Δ
write(2)	1		

SystemC



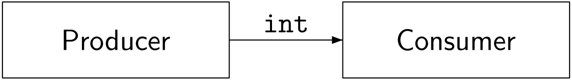
Producer	signal	Consumer	
Kernel: make all processes active			
skip	–	`wait(value_changed)`	start Δ
Kernel: make Producer active			end Δ
write(1)	notify(value_changed)	sleeping	start Δ
Kernel: update value of signal			update
Kernel: Make Producer and Consumer active			end Δ
write(2)	1	read(1)	

SystemC



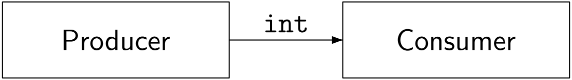
Producer	signal	Consumer	
Kernel: make all processes active			
skip	–	`wait(value_changed)`	start Δ
Kernel: make Producer active			end Δ
write(1)	notify(value_changed)	sleeping	start Δ
Kernel: update value of signal			update
Kernel: Make Producer and Consumer active			end Δ
write(2)	1	read(1)	
Kernel: Make Producer and Consumer active			

SystemC



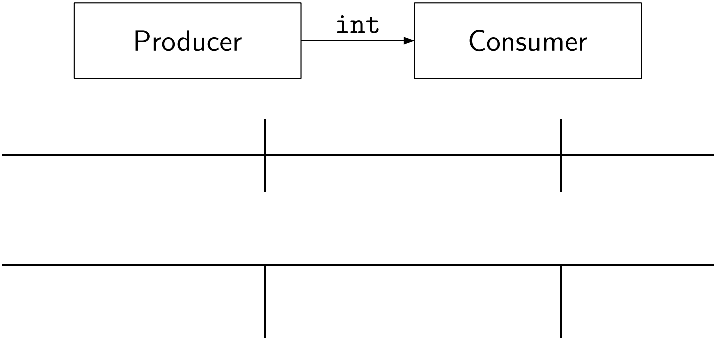
Producer	signal	Consumer	
Kernel: make all processes active			
skip	–	`wait(value_changed)`	start Δ
Kernel: make Producer active			end Δ
write(1)	notify(value_changed)	sleeping	start Δ
Kernel: update value of signal			update
Kernel: Make Producer and Consumer active			end Δ
write(2)	1	read(1)	start Δ
Kernel: Make Producer and Consumer active			end Δ

SystemC

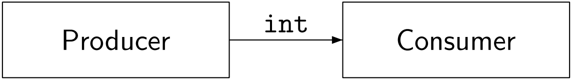


Producer	signal	Consumer	
Kernel: make all processes active			
skip	–	`wait(value_changed)`	start Δ
Kernel: make Producer active			end Δ
write(1)	notify(value_changed)	sleeping	start Δ
Kernel: update value of signal			update
Kernel: Make Producer and Consumer active			end Δ
write(2)	1	read(1)	start Δ
Kernel: Make Producer and Consumer active			end Δ
. . .	. . .	. . .	

SystemC

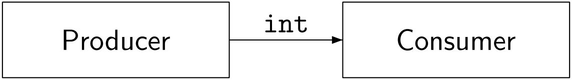


SystemC



Producer	signal	Consumer

SystemC



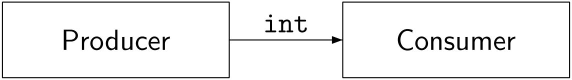
Producer

int

Consumer

Producer	signal	Consumer
write(1);		

SystemC

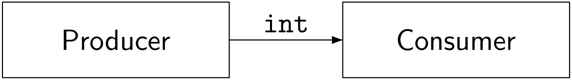

Producer

`int`

Consumer

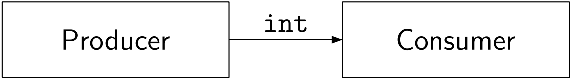
Producer	signal	Consumer
write(1); *write(2)*;		

SystemC



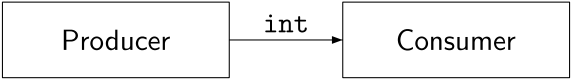
Producer	signal	Consumer
write(1); *write(2)*;	notify(value_changed)	

SystemC



Producer	signal	Consumer
write(1); *write(2)*;	notify(value_changed)	wait()

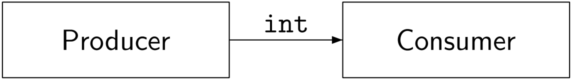
SystemC



Producer	signal	Consumer
write(1); *write(2)*;	notify(value_changed)	wait()

Kernel: update value of signal

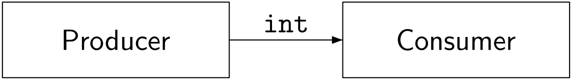
SystemC


Producer
int
Consumer

Producer	signal	Consumer
write(1); *write(2)*;	notify(value_changed)	wait()

Kernel: update value of signal

Kernel: Make Producer and Consumer active



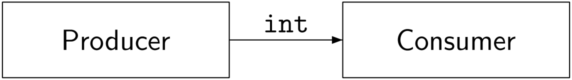
Producer	signal	Consumer
write(1); *write(2)*;	notify(value_changed)	wait()

Kernel: update value of signal

Kernel: Make Producer and Consumer active

skip		

SystemC

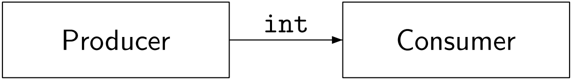

Producer

int

Consumer

<table>
<tr><th>Producer</th><th>signal</th><th>Consumer</th></tr>
<tr><td>write(1); <i>write(2)</i>;</td><td>notify(value_changed)</td><td>wait()</td></tr>
<tr><td colspan="3"><i>Kernel: update value of signal</i></td></tr>
<tr><td colspan="3"><i>Kernel: Make Producer and Consumer active</i></td></tr>
<tr><td>skip</td><td><i>2</i></td><td></td></tr>
</table>

SystemC



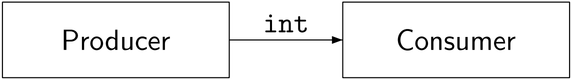
Producer	signal	Consumer
write(1); *write(2)*;	notify(value_changed)	wait()

Kernel: update value of signal

Kernel: Make Producer and Consumer active

skip	*2*	read(2)

SystemC



<table>
<tr><th>Producer</th><th>signal</th><th>Consumer</th></tr>
<tr><td>write(1); write(2);</td><td>notify(value_changed)</td><td>wait()</td></tr>
<tr><td colspan="3">Kernel: update value of signal</td></tr>
<tr><td colspan="3">Kernel: Make Producer and Consumer active</td></tr>
<tr><td>skip</td><td>2</td><td>read(2)</td></tr>
<tr><td>...</td><td>...</td><td>...</td></tr>
</table>

Desideratum: Express properties at sub-Δ-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

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

Key observation: No canonical notion of a cycle

“Useful” SystemC properties: examples

- A signal is written to at most once
- The value of variable “balance” is always equal to “deposits” - “withdrawals”
- “Request” $\rightarrow$ within[3] “grant”

“Useful” SystemC properties: examples

- 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” $\longrightarrow$ within[3] “grant”

“Useful” SystemC properties: examples

- 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”

“Useful” SystemC properties: examples

- 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” $\rightarrow$ within[3] “grant”
  - Within 3 “what”?

Temporal language for SystemC: Desiderata

- 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

A New Approach

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

Dealing with the kernel

Why deal with the kernel?

- Many important properties at sub-Δ cycle resolution
- Adapt specifications to level of abstraction

Example: Invariance properties, say, `ALWAYS x > 10`

- Must hold at *all* times
- Must hold when processes suspend
- Must hold at delta-cycle boundary
- Must hold at clock-cycle boundary

This is possible *only* if we require the kernel to expose information about its internal state.

Dealing with the kernel

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

Dealing with the kernel

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

Code (Consumer.h)

```
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)

```
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`)

Traces

- A SystemC trace is a sequence of states corresponding to execution of model
- Expose alternation of control
    - kernel
    - user code
    - libraries
- “large-step semantics” vs “small-step semantics”
    - `y = (x++) + (x--);`

Refining specifications

- Balance = Deposits - Withdrawals

Refining specifications

- Balance = Deposits - Withdrawals
    - At end of all Δ cycles

Refining specifications

- Balance $=$ Deposits - Withdrawals
  - At end of all Δ cycles
  - When no process is running

Refining specifications

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

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

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

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

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

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 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
  - Only once

Software Verification Perspective

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

Software Verification Perspective

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

Conclusion and Discussion

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

Appendix

Appendix

Semantics of SystemC Simulation I

```
$PC \leftarrow$ all primitive channels
$P \leftarrow$ all processes
$R \leftarrow P$ /* Set of runnable processes */
$D \leftarrow \emptyset$ /* Set of pending delta notifications */
$U \leftarrow \emptyset$ /* Set of update requests */
$T \leftarrow \emptyset$ /* Set of pending timed notifications */
for all $chan \in PC$ do
    run $chan$.update()
for all $p \in R$ do
    if $p$ is initializable then
        run $p$
for all $d \in D$ do
    $D \leftarrow D \setminus d$
    for all $p \in P$ do
        if $n$ triggers $p$ then
            $R \leftarrow R \cup p$
```

Semantics of SystemC Simulation II

```
repeat
  while $R \neq \emptyset$ do /* New delta cycle begins */
    for all $r \in R$ do
      $R \leftarrow R \setminus r$
      run $r$ until it invokes wait() or returns
    for all $chan \in U$ do /* Update phase */
      run $chan$.update()
    for all $d \in D$ do /* Delta notification phase */
      $D \leftarrow D \setminus d$
      for all $p \in P$ do
        if $d$ triggers $p$ then
          $R \leftarrow R \cup p$ /* $p$ is now runnable */
  /* End of delta cycle */
```

Semantics of SystemC Simulation III

```
  if $T \neq \emptyset$ then
    Advance clock to earliest timed delay $t$.
    $T \leftarrow T \setminus t$
    for all $p \in P$ do
      if $t$ triggers $p$ then
        $R \leftarrow R \cup p$ /* p is now runnable */
until end of simulation
```

Semantics of SystemC Simulation - Kernel Phase

Our approach: keep track of current phase

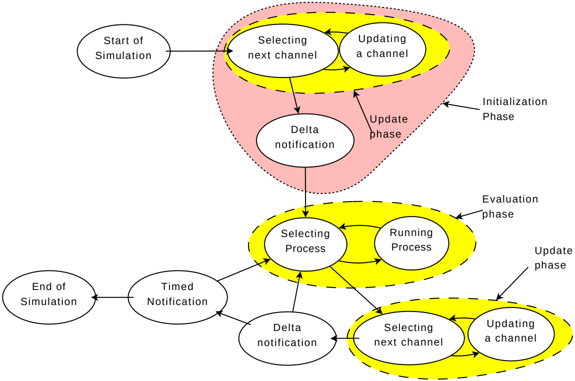


Figure: Captured Kernel States

Kernel States in Moy's Abstraction

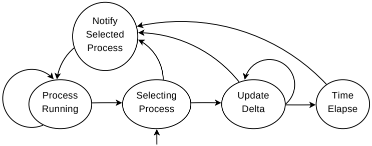


Figure: Kernel states proposed by Moy et al.