



The SystemC Simulation Engine

An Interactive Exploration
of an Event Driven Simulator

Doulos Inc.

David C. Black, Senior Member Technical Staff

1



Copyright Permissions

- This work is copyright 2013-2018 by Doulos Inc. with all rights reserved. Permission is hereby granted for professors of recognized licensed Colleges and Universities to use in lectures including the creation of handouts for registered students. Individuals may download and use for personal education, but not for display, public presentation nor redistribution electronically or in any other format. Any other commercial use is strictly prohibited without explicit written permission from Doulos Inc.
- In simpler language, competitors to Doulos training services may not use this material unless separately licensed. Display on the Internet is prohibited except from Doulos authorized locations.

2



Agenda & Goals

- Overview of flow diagram and queues
- SystemC constructs used
- [Example code](#)
- [Step-by-step walk-thru](#)
 - Illustrate each step of simulation
 - Understand events & delta cycles
- References

Version 2.4 – © Doulos 2013-2019 – All Rights Reserved

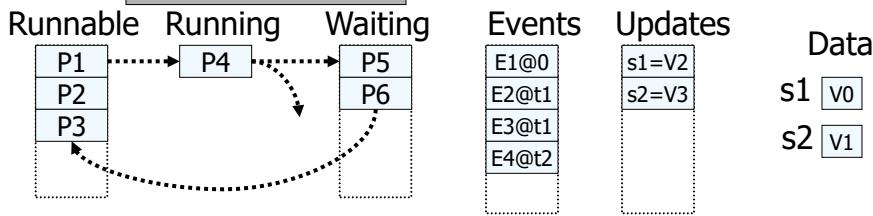
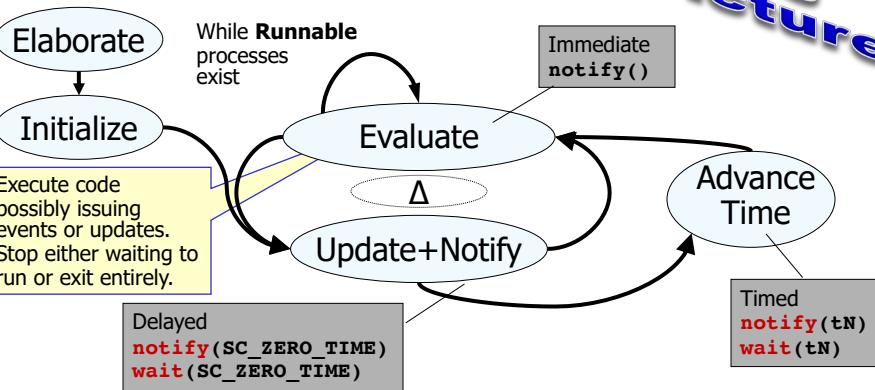
3



Simulation Engine

Big Picture

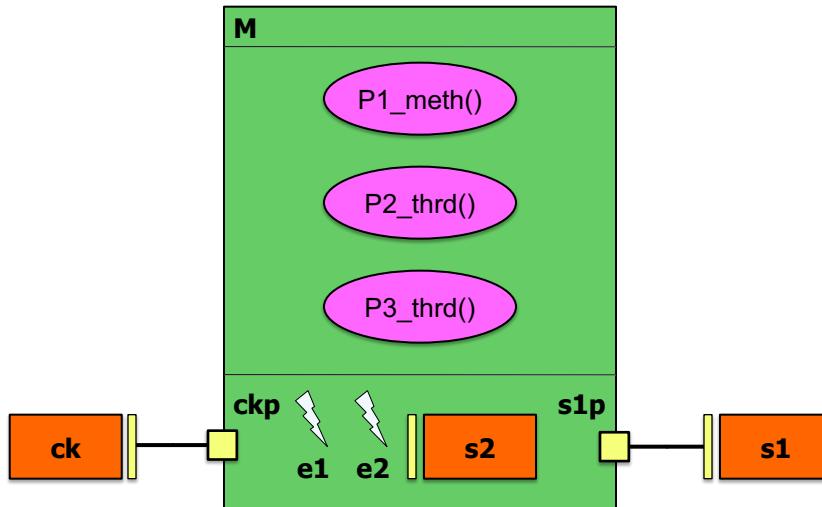
Version 2.4 – © Doulos 2013-2019 – All Rights Reserved



4

Example Design

Version 2.4 – © Doulos 2013-2019 – All Rights Reserved



5

SystemC constructs used herein

Version 2.4 – © Doulos 2013-2019 – All Rights Reserved

- **SC_MODULE, SC_CTOR** - used to create modules
- **SC_THREAD¹, SC_METHOD¹** - types of processes
- **sensitive, dont_initialize** - attributes of processes
- **sc_event, wait², notify³** - synchronization mechanisms
- **sc_signal⁴, read, write** - primitive channel
- **sc_clock - sc_signal<bool>** with a generating process
- **sc_in - sc_port<>** specialization of type **sc_signal_in_if<>**
- **sc_out - sc_port<>** specialization of type **sc_signal_out_if<>**

¹ Verilog **initial** or **always** block; VHDL **process** block

² Verilog **@**, **#**, or **wait** statement; VHDL **wait** statement

³ Verilog **->** statement, except SystemC is more flexible

⁴ Verilog **wire** or **var** with **<=** type; VHDL **signal** type

6



Example SystemC

Version 2.4 – © Doulos 2013-2019 – All Rights Reserved

```
sc_clock ck("ck",6,0.5,3);
SC_MODULE(M) {
    sc_in<bool> ckp;//in port
    sc_out<int> s1p;//out port
    sc_signal<int> s2;
    sc_event e1, e2;
    void P1_meth();
    void P2_thrd();
    void P3_thrd();
    SC_CTOR(M):temp(9){
        SC_THREAD(P3_thrd);
        {
            SC_THREAD(P2_thrd);
            sensitive<<ckp.pos();
        }
        {
            SC_METHOD(P1_meth);
            sensitive<<s2;
            dont_initialize();
        }
    } //end SC_CTOR
private:
    int temp;
};
```

Executed during elaboration

```
void M::P1_meth() {
    temp = s2.read();
    s1p->write(temp+1);
    e2.notify(2,SC_NS); //timed
}
void M::P2_thrd() {
A: s2.write(5);
    e1.notify(); //immediate
    wait();
B: for (int i=7;i<9;i++){
    s2.write(i);
    wait(1,SC_NS);
} //delayed
C: e1.notify(SC_ZERO_TIME);
    wait(); //static sensitive
} //endfor
}
void M::P3_thrd() {
D: while(true) {
    wait(e1|e2);
E: cout << "time "
    <<sc_time_stamp()<<endl;
} //endwhile
}
```

Simulation



Example SystemVerilog

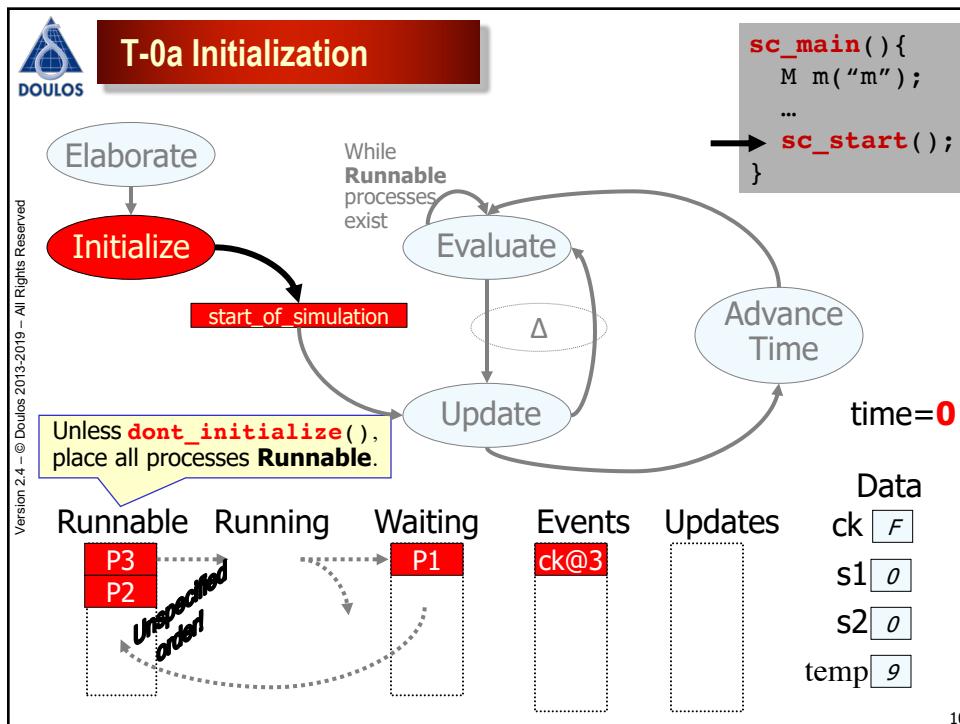
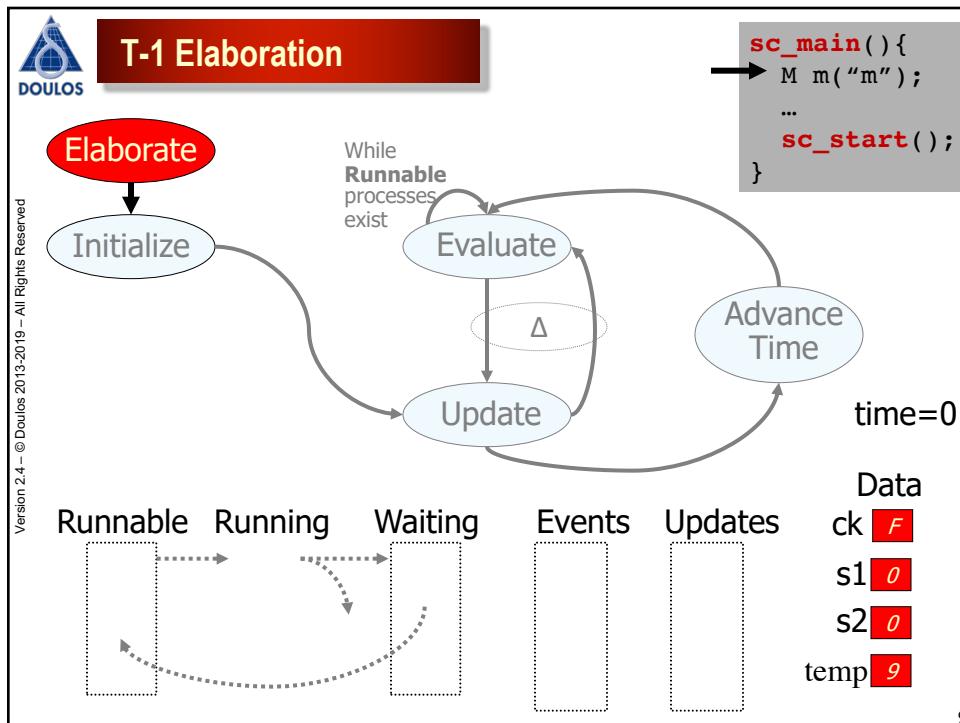
Version 2.4 – © Doulos 2013-2019 – All Rights Reserved

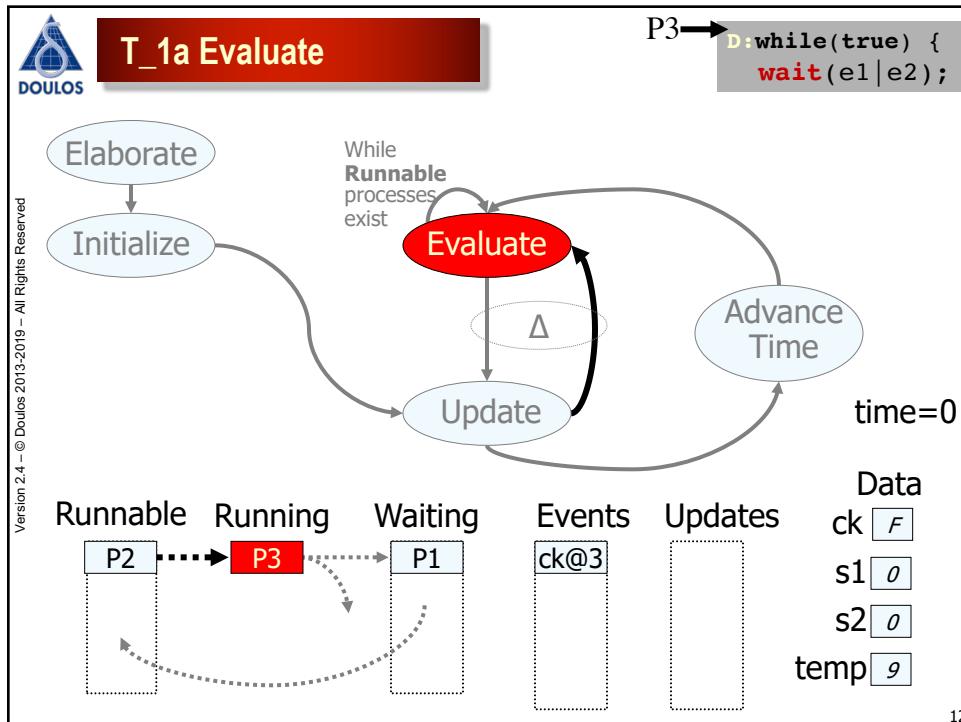
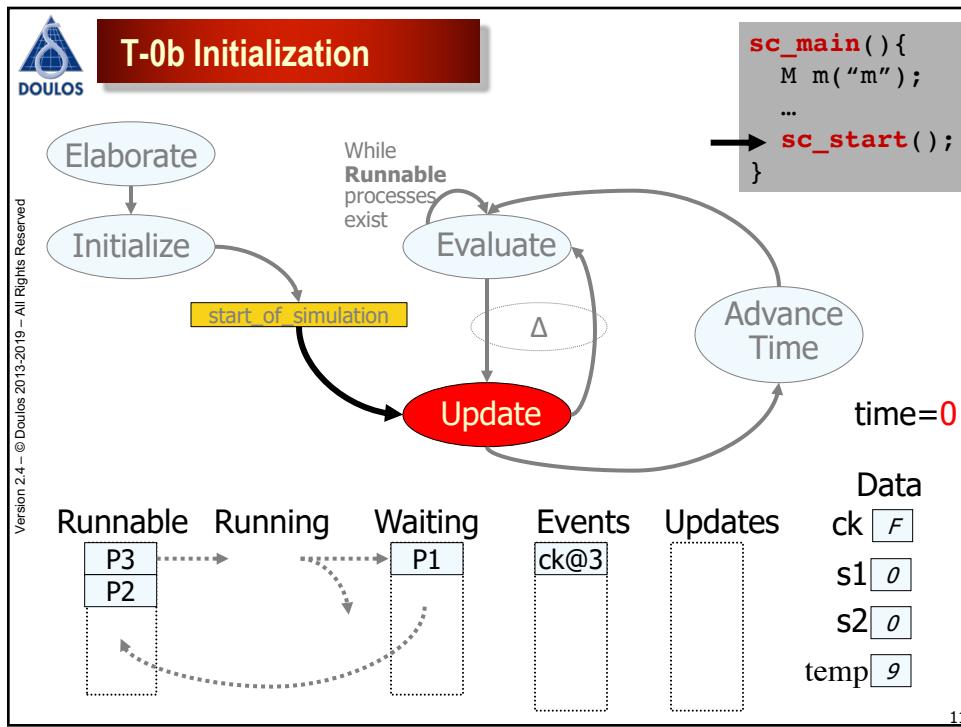
```
module M
    int temp = 9;
    bit ck; always #3 ck++;
    int s1p;//out port
    int s2;
    event e1, e2;
endmodule
```

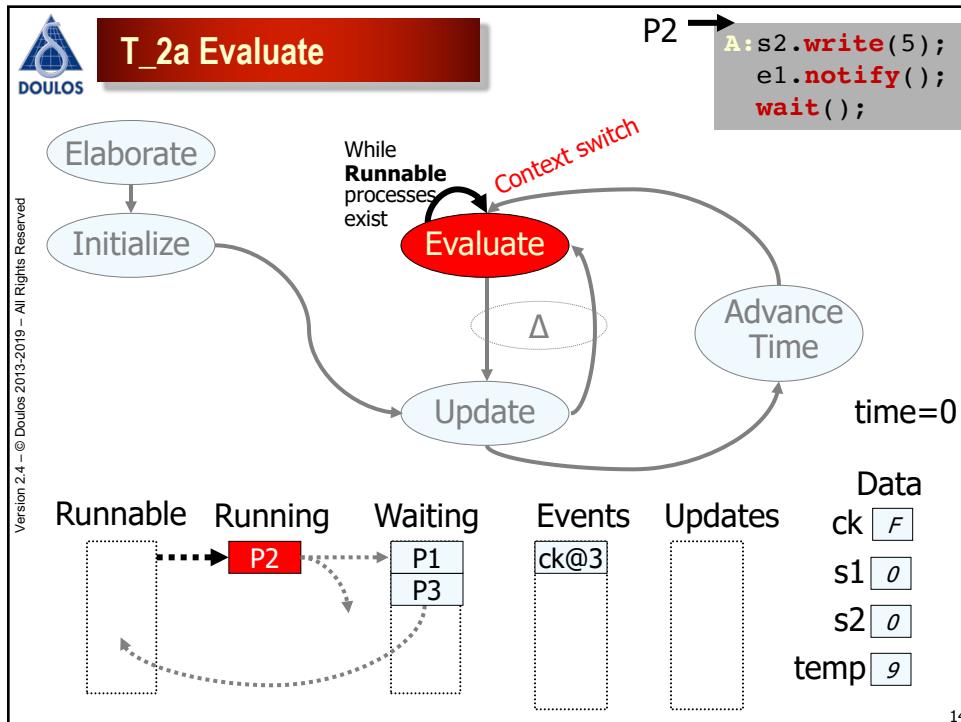
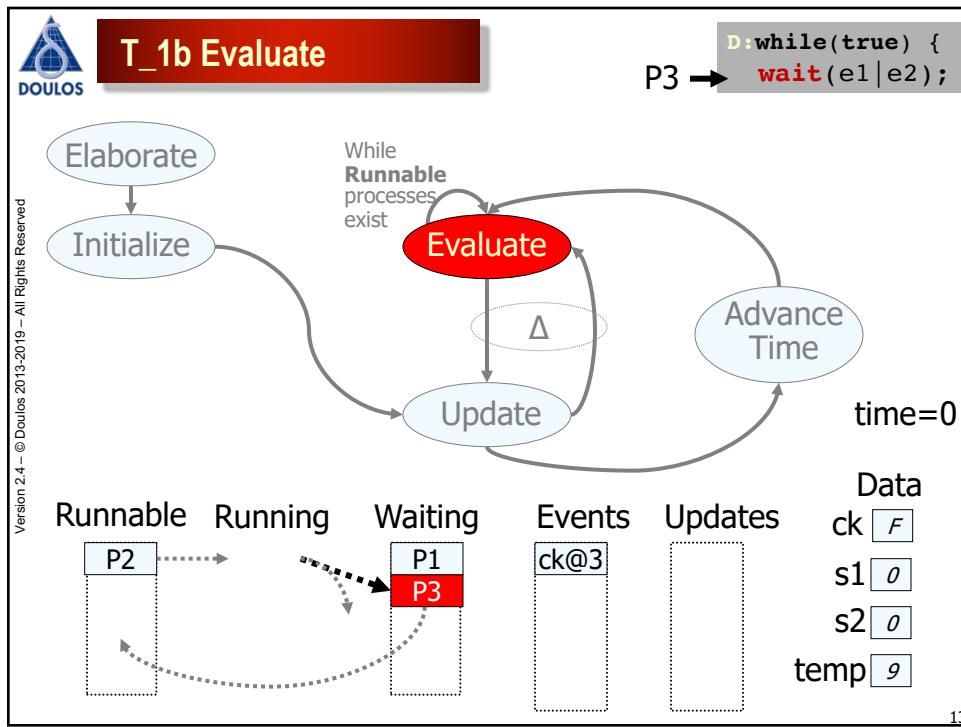
Executed during elaboration

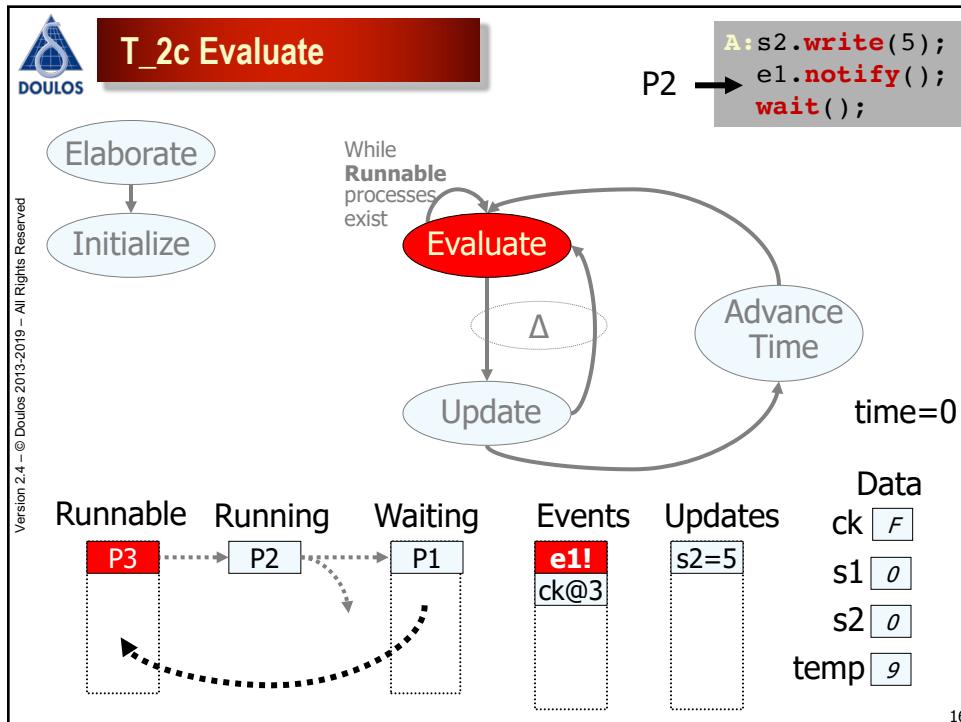
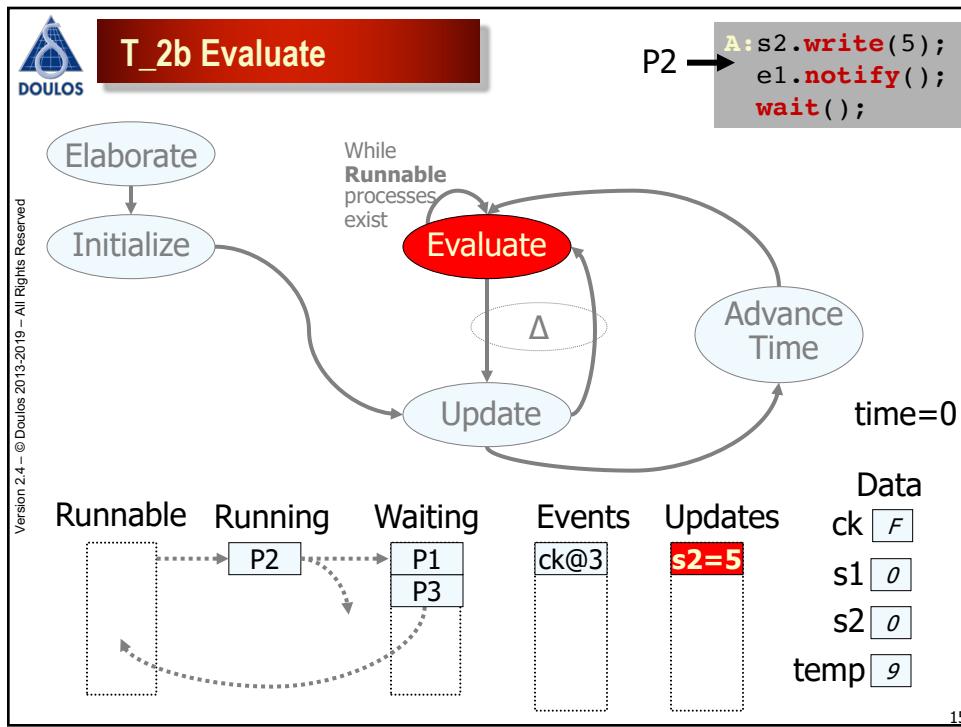
```
P1_meth: always @ (s2) begin
    temp = s2;
    s1p <= temp+1;
    ->>#2ns e2; //timed
end
P2_thrd: initial begin
A: s2 <= 5;
    ->>e1; //immediate
    @(posedge ck);
B: for (int i=7;i<9;i++) begin
    s2 <= i;
    #1ns;
C: ->>#0 e1; //delayed
    @(posedge ck);
end
end
P3_thrd: initial begin
D: while(1) {
    @(e1 or e2);
E: $display("time %t",$time);
    end: D
} //delayed
end
```

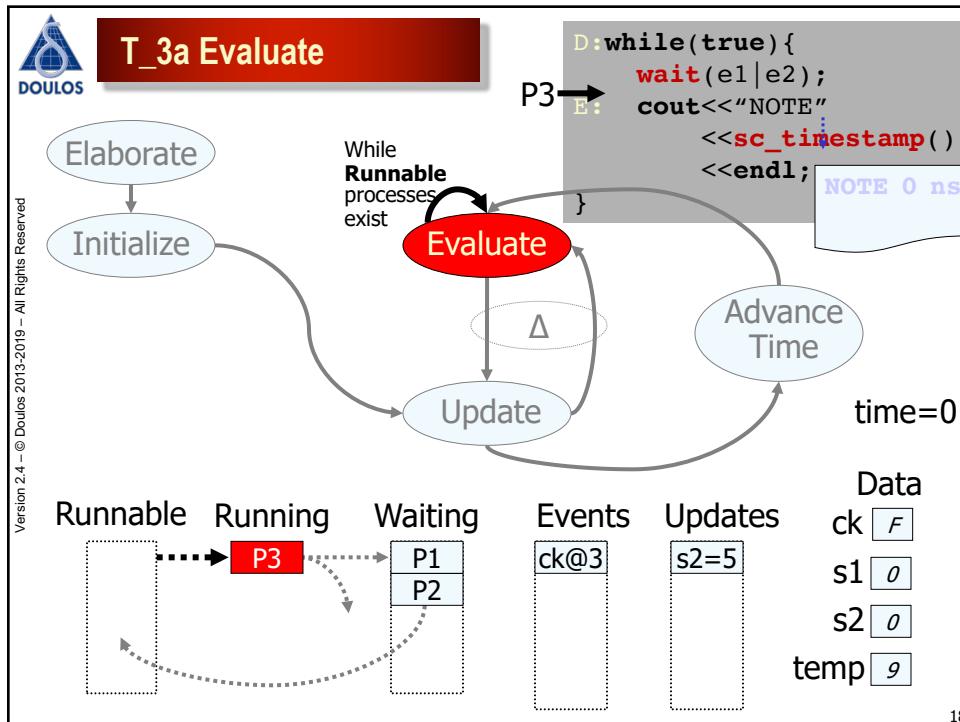
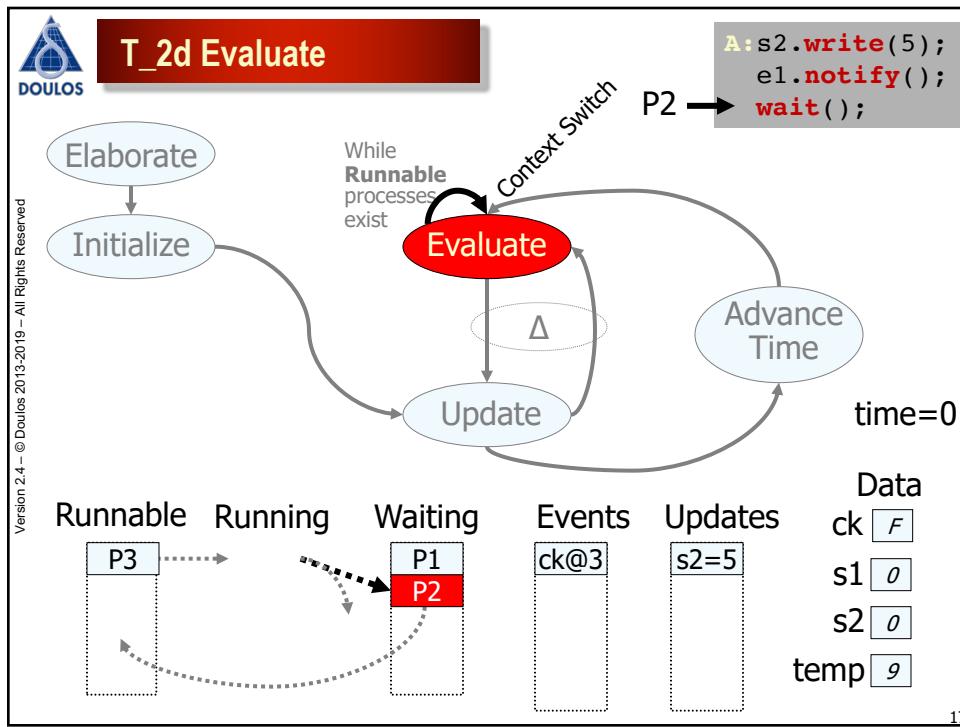
Simulation

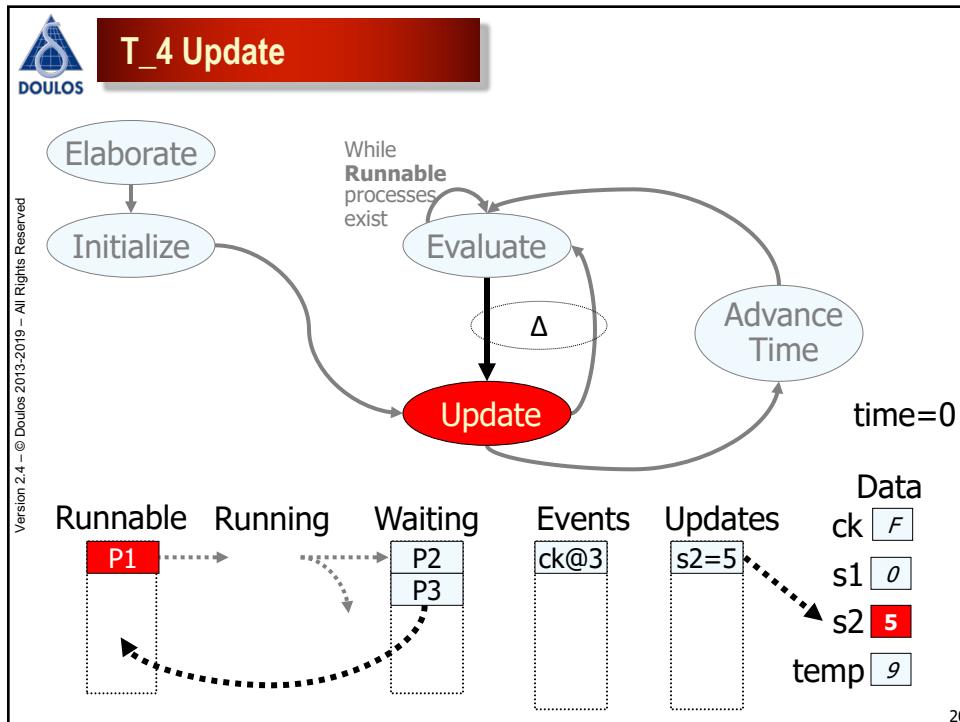
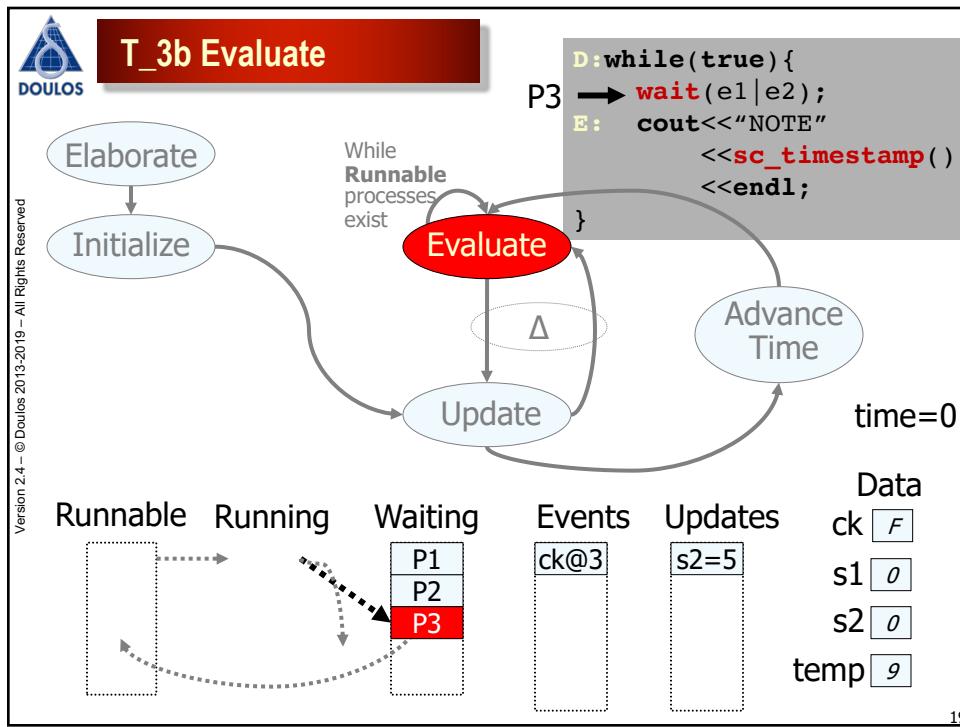


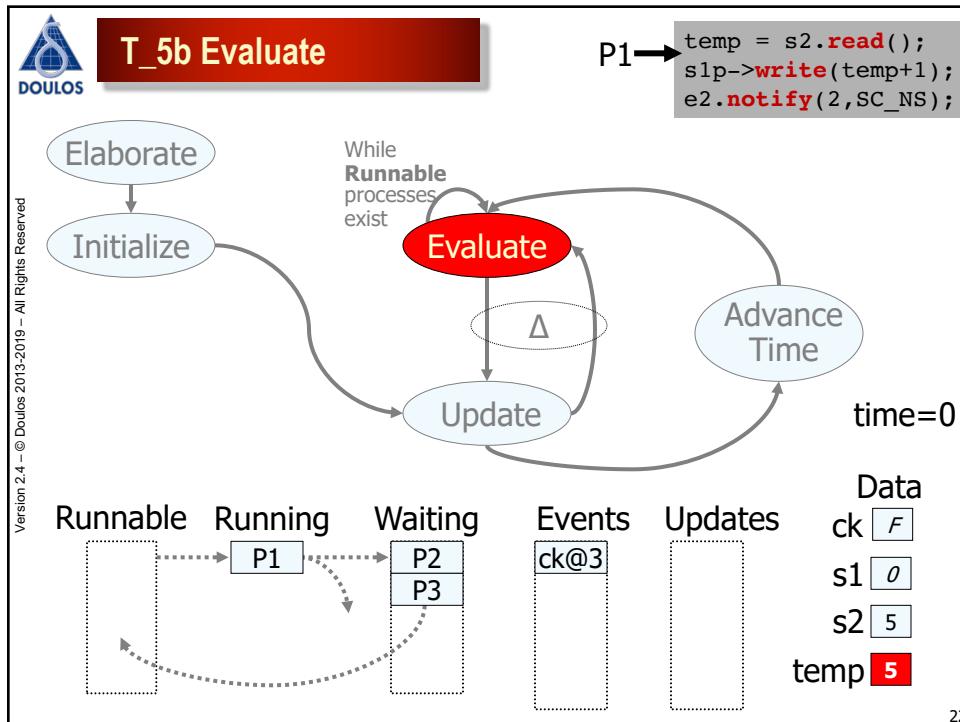
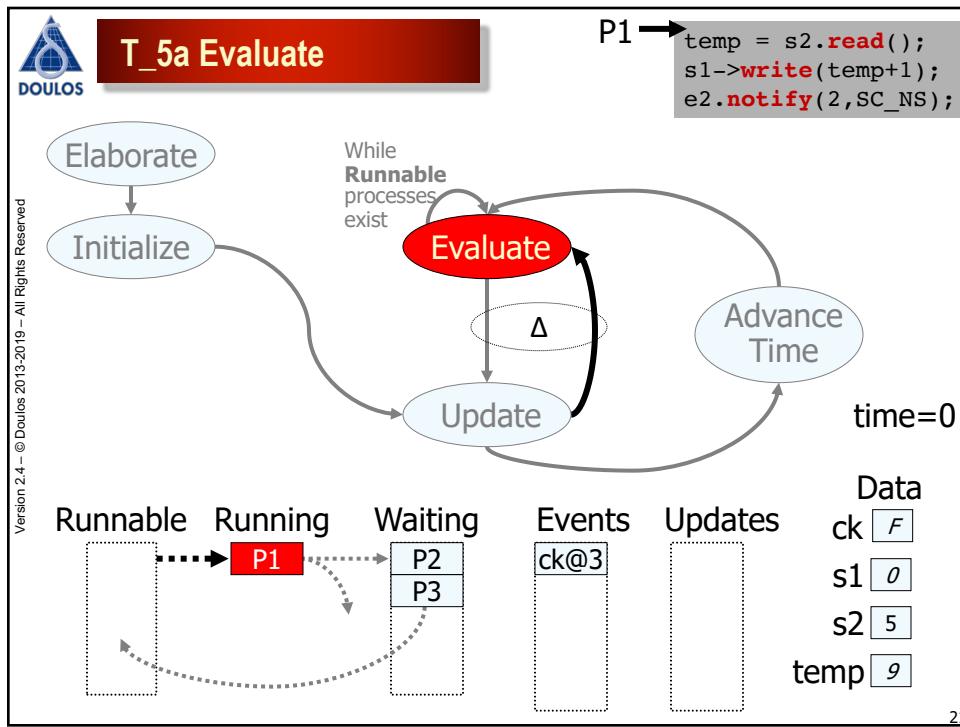


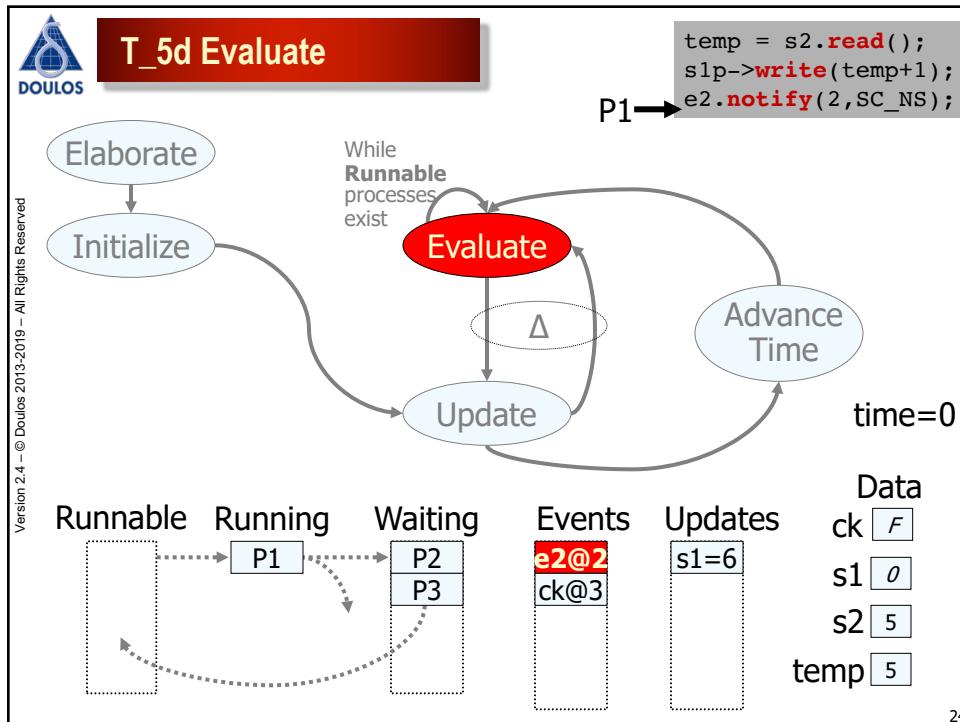
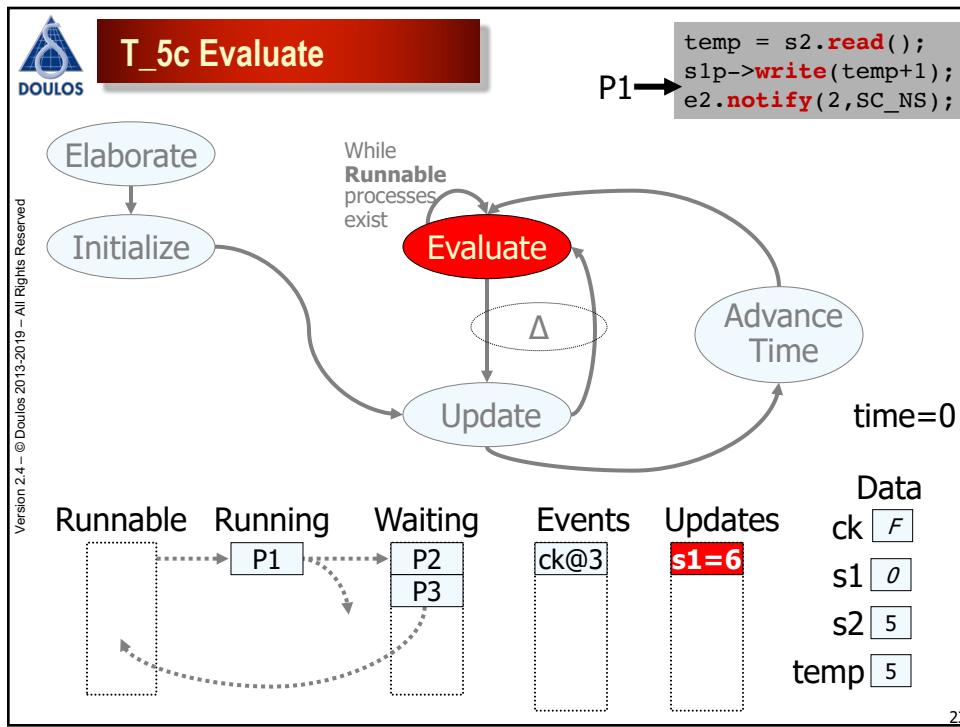


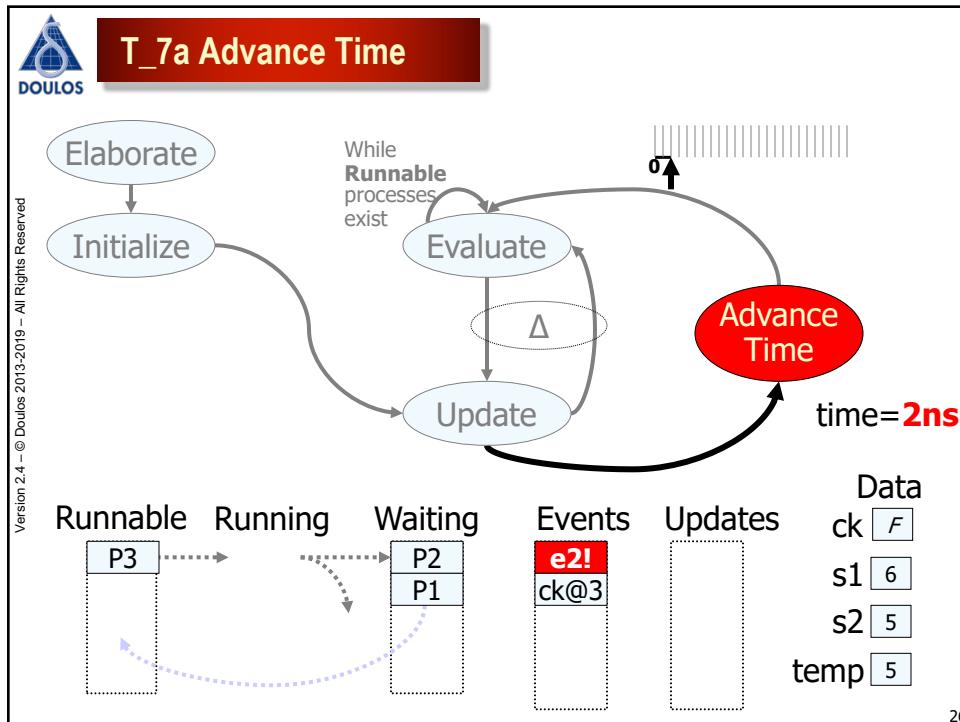
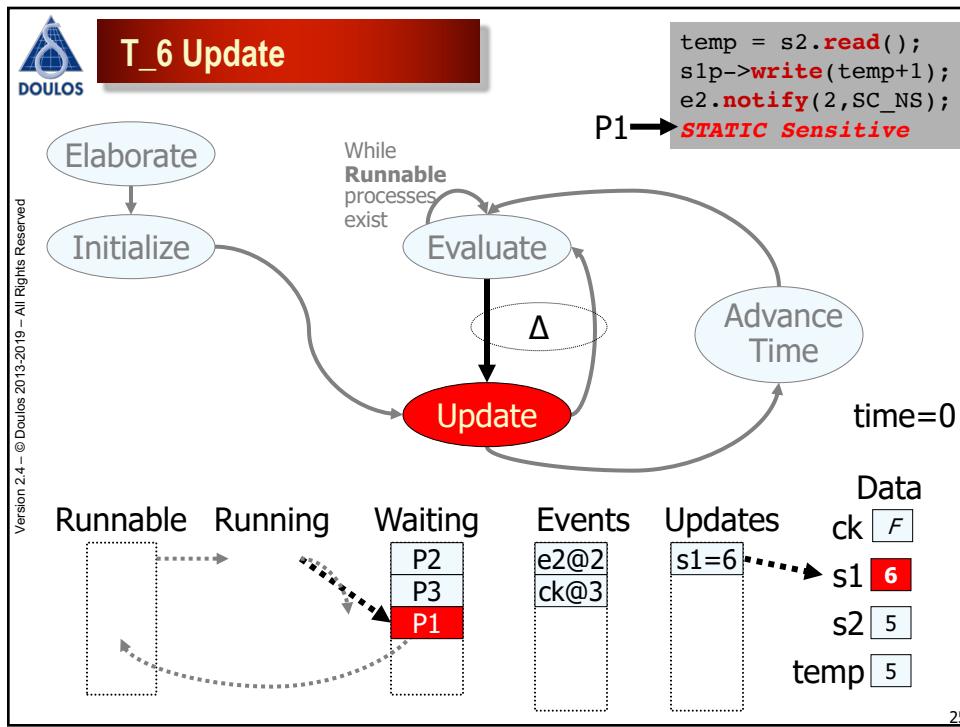


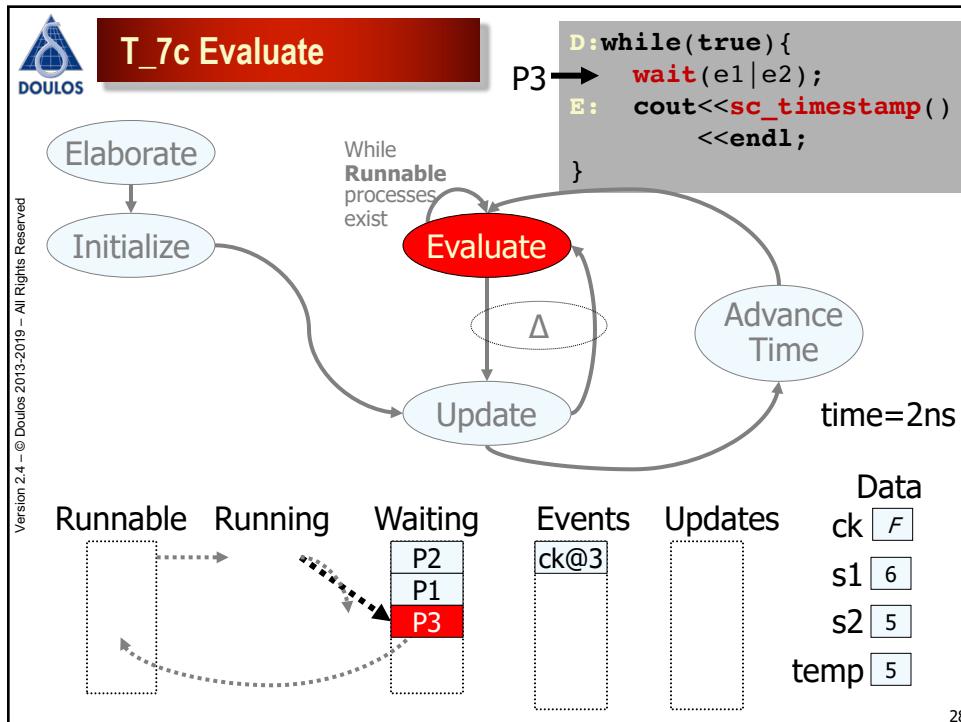
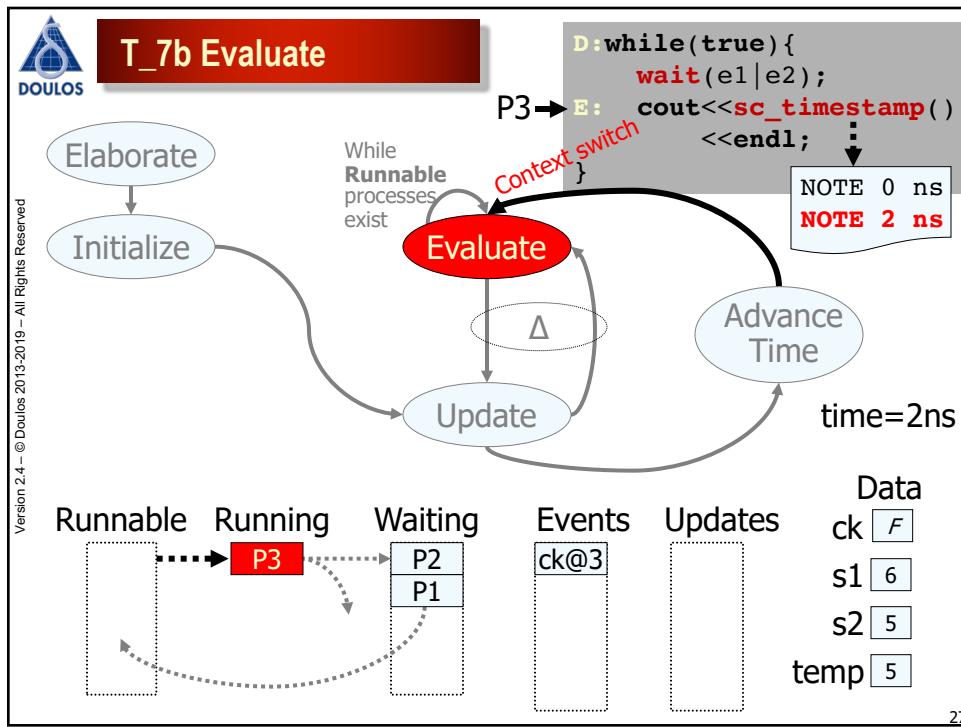


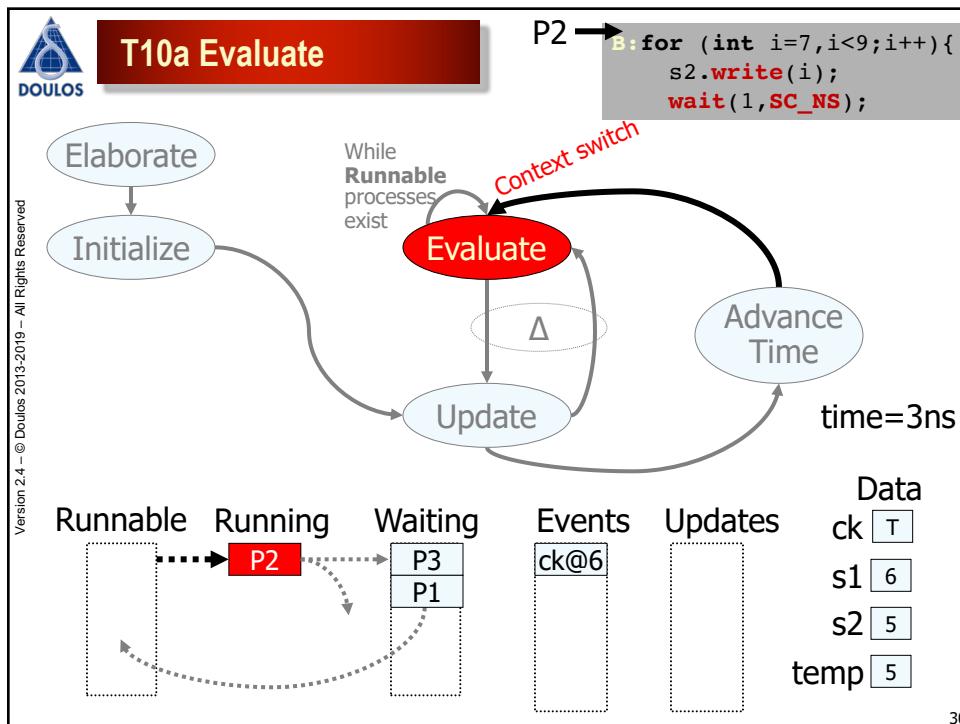
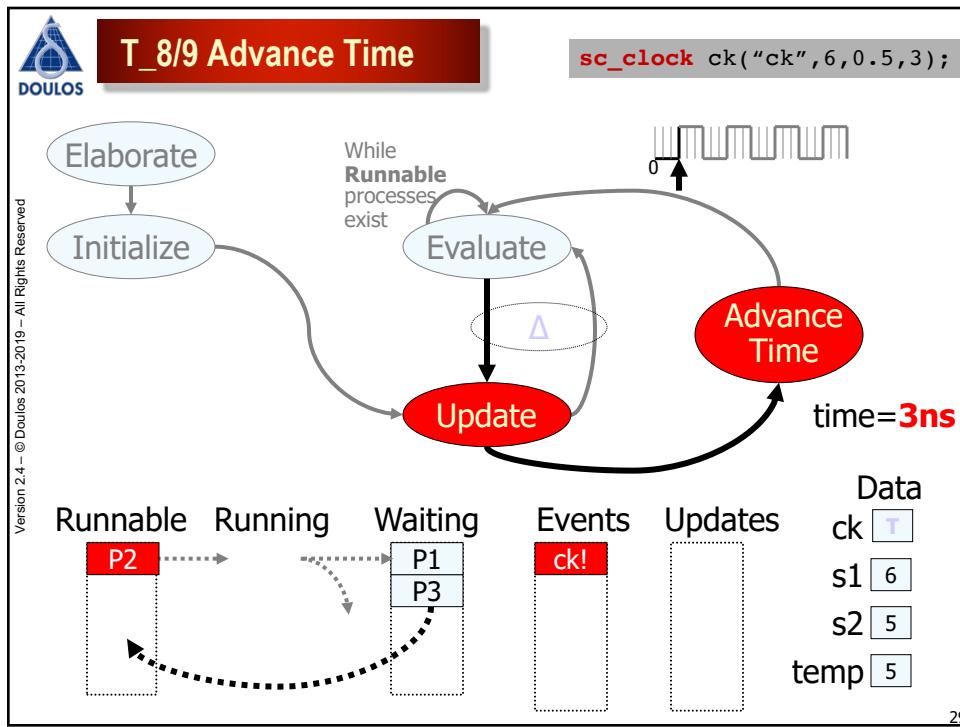


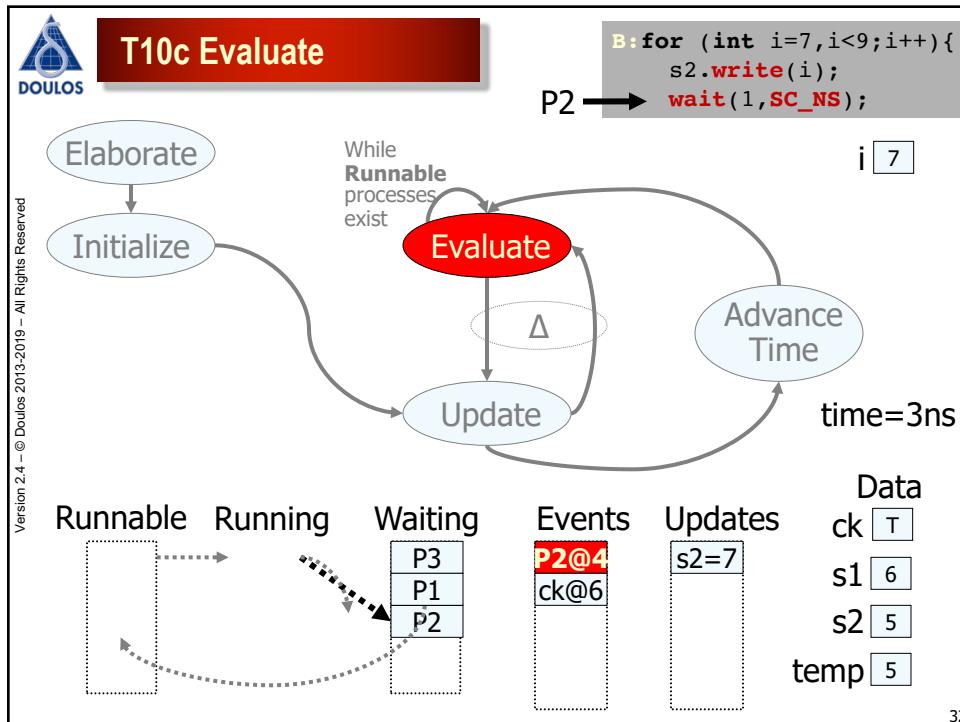
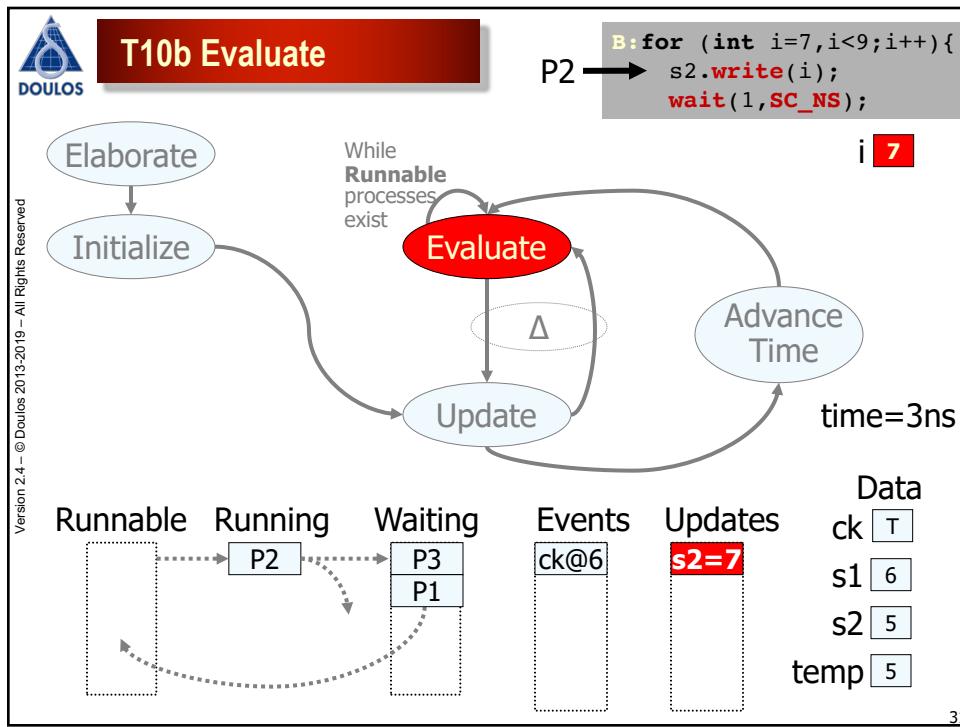


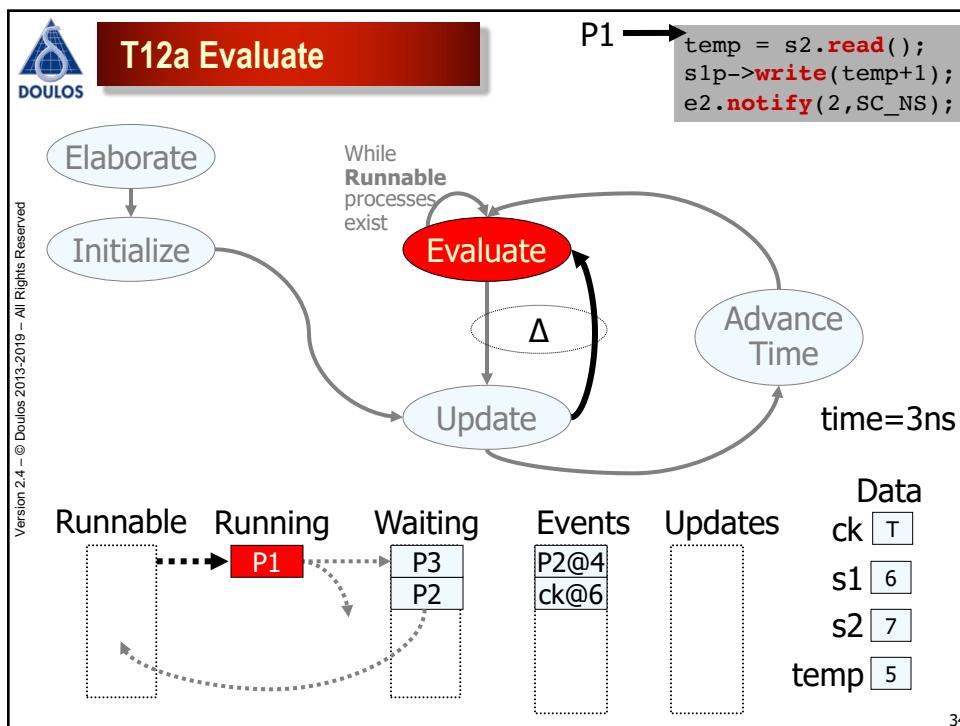
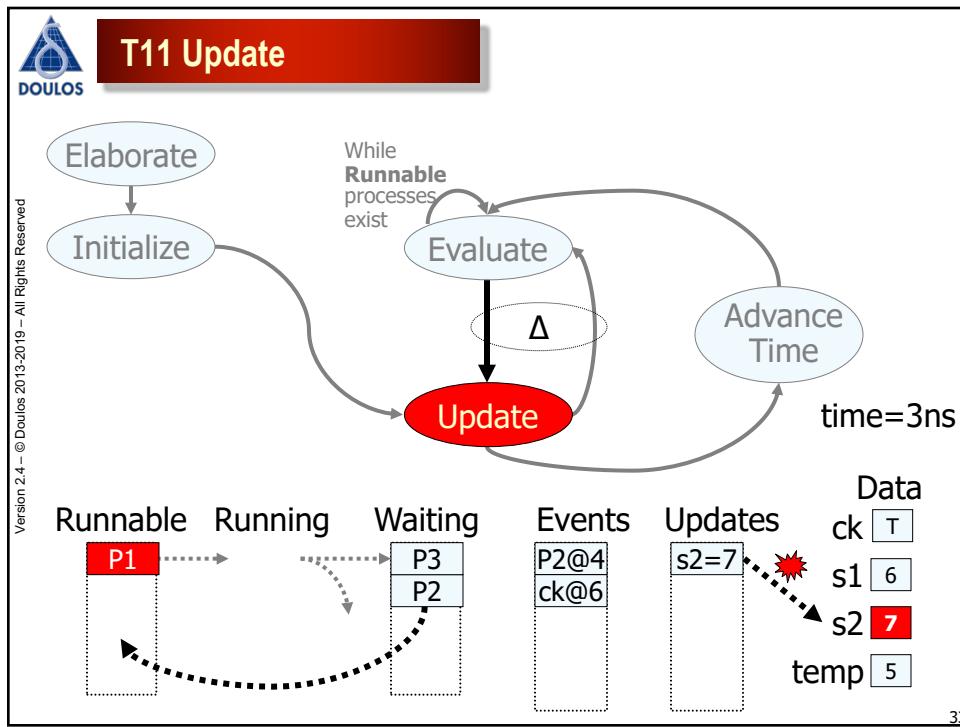


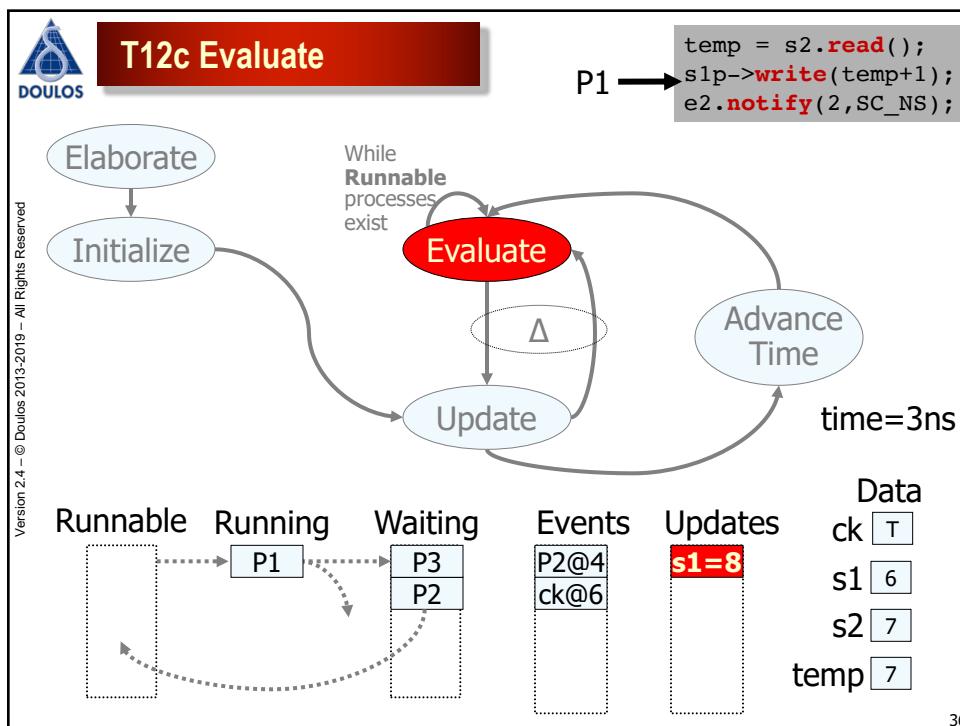
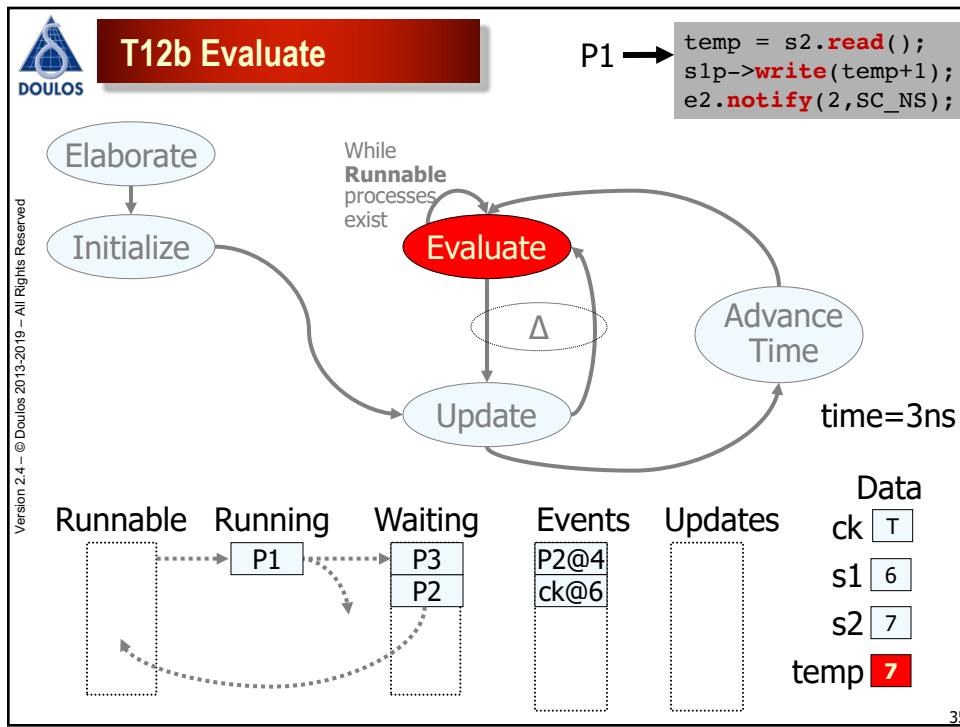


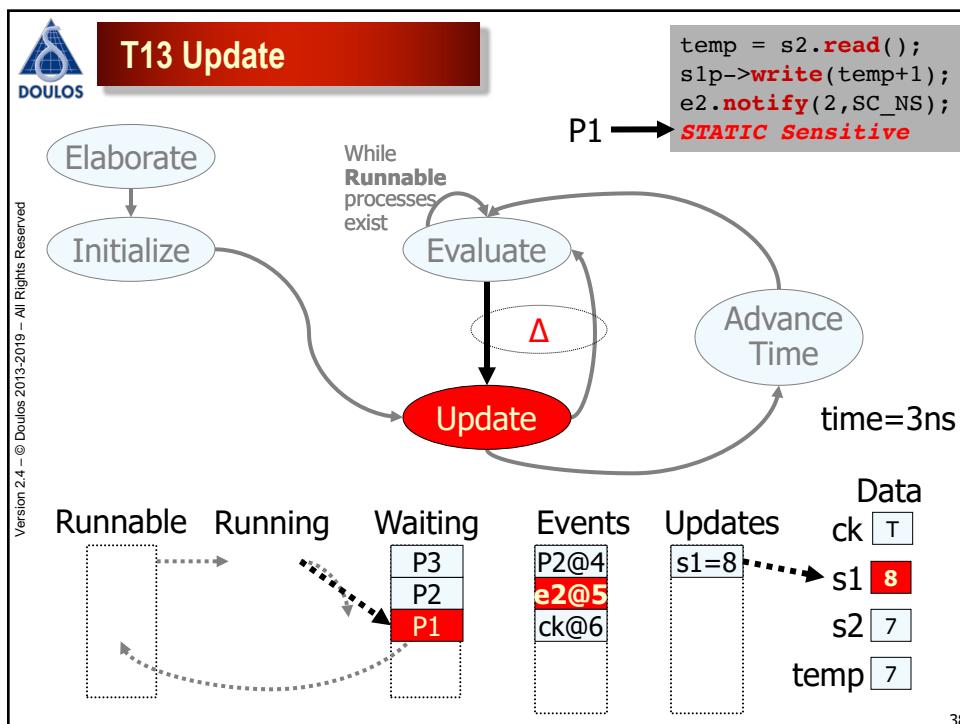
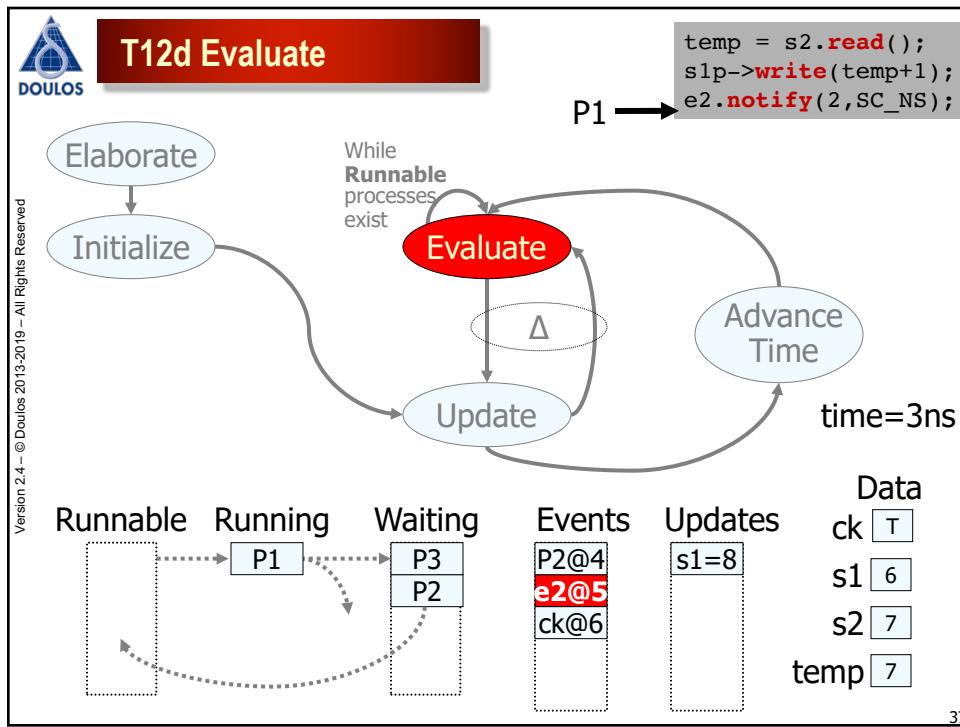


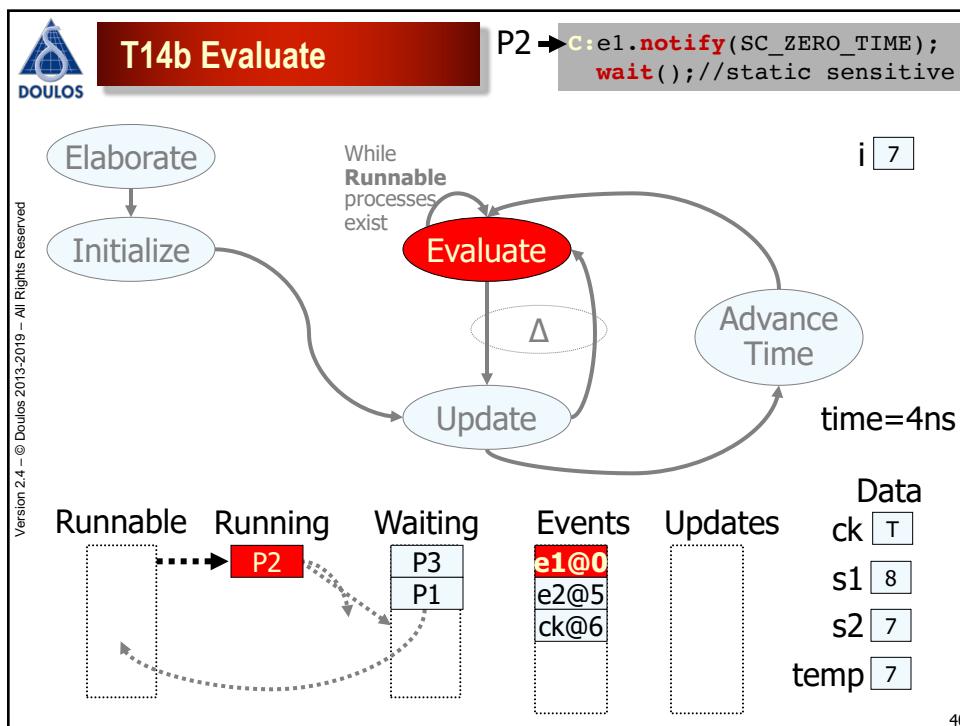
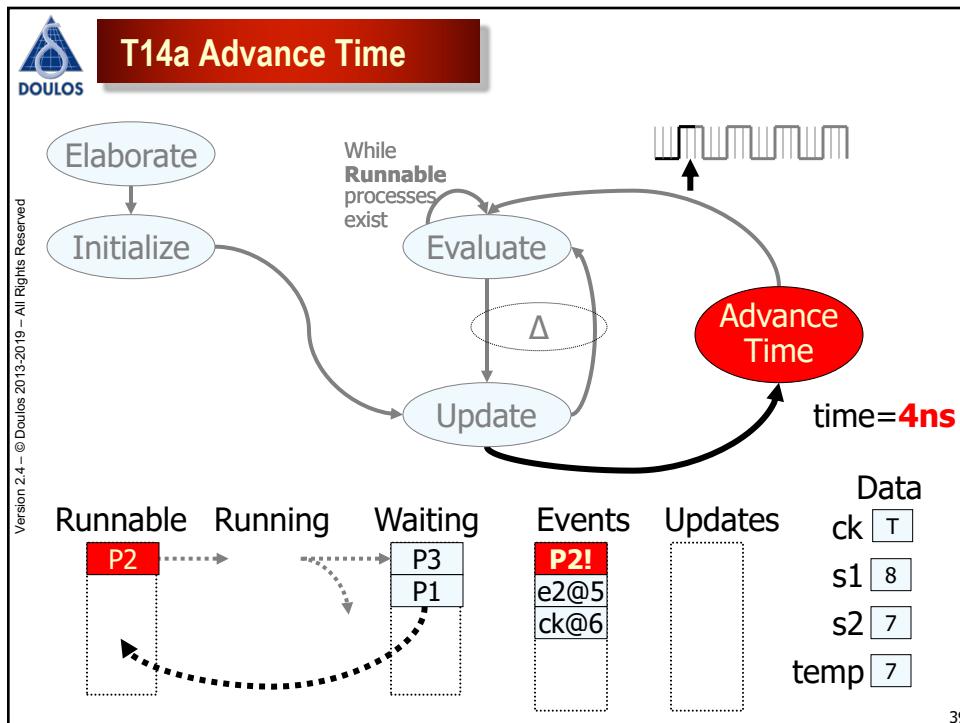


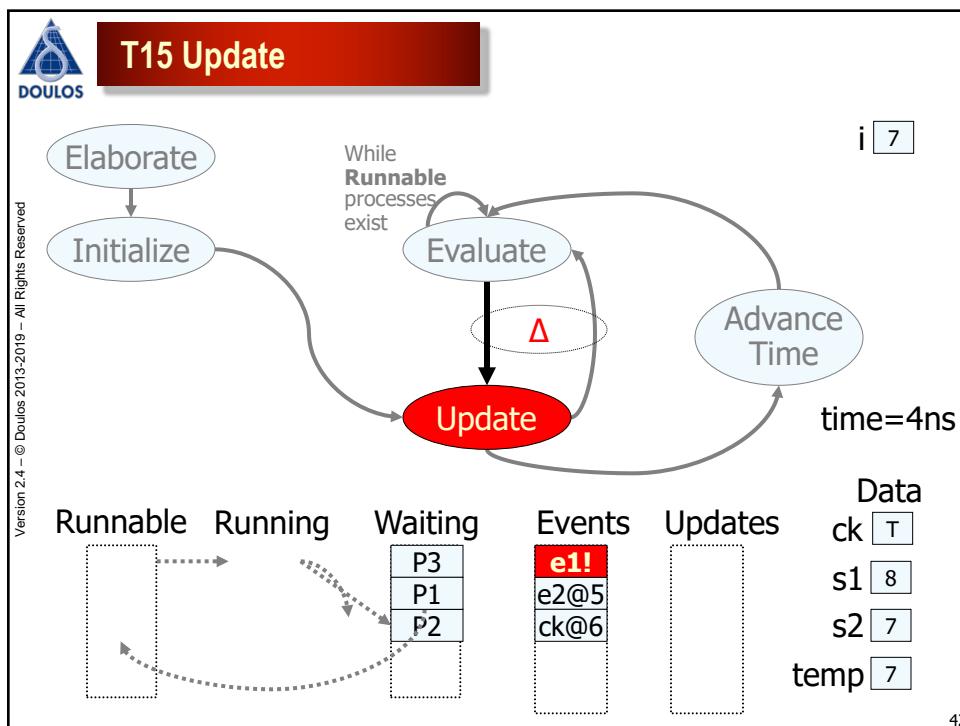
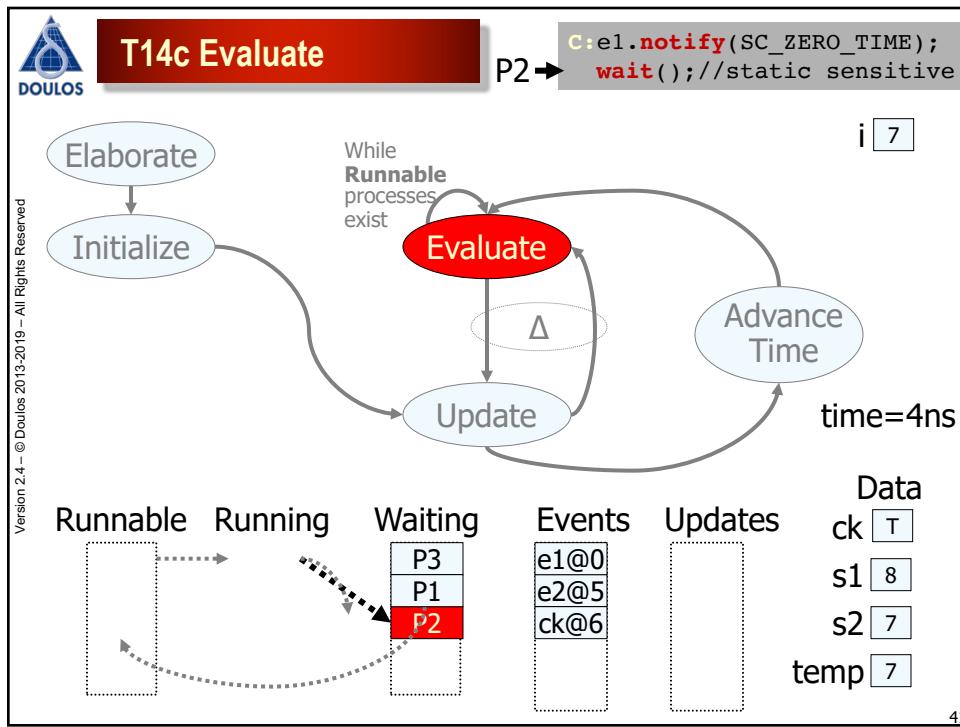


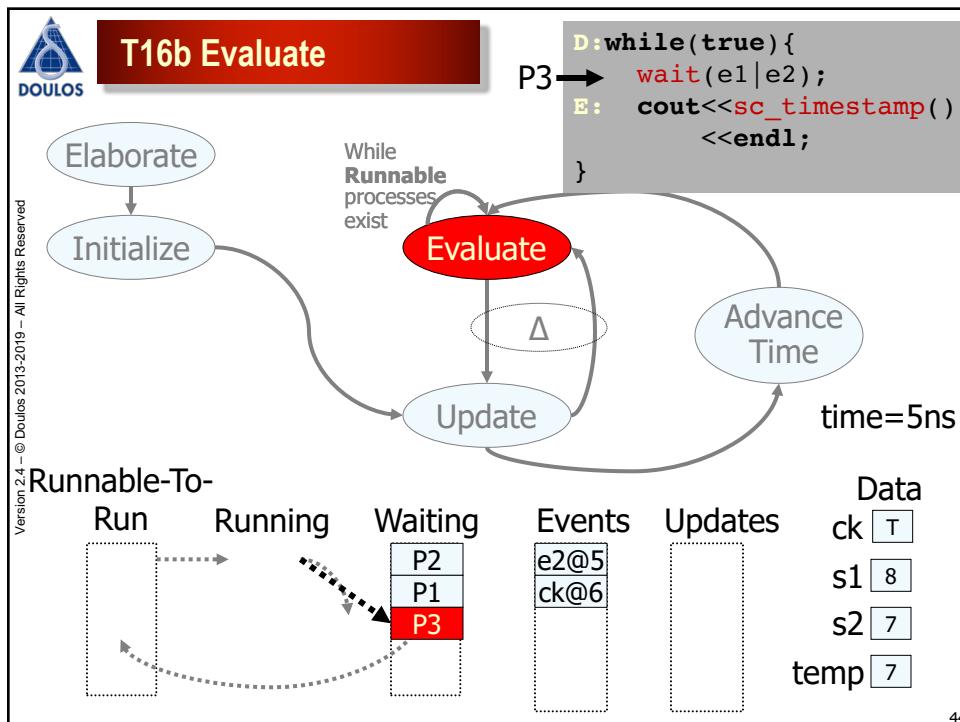
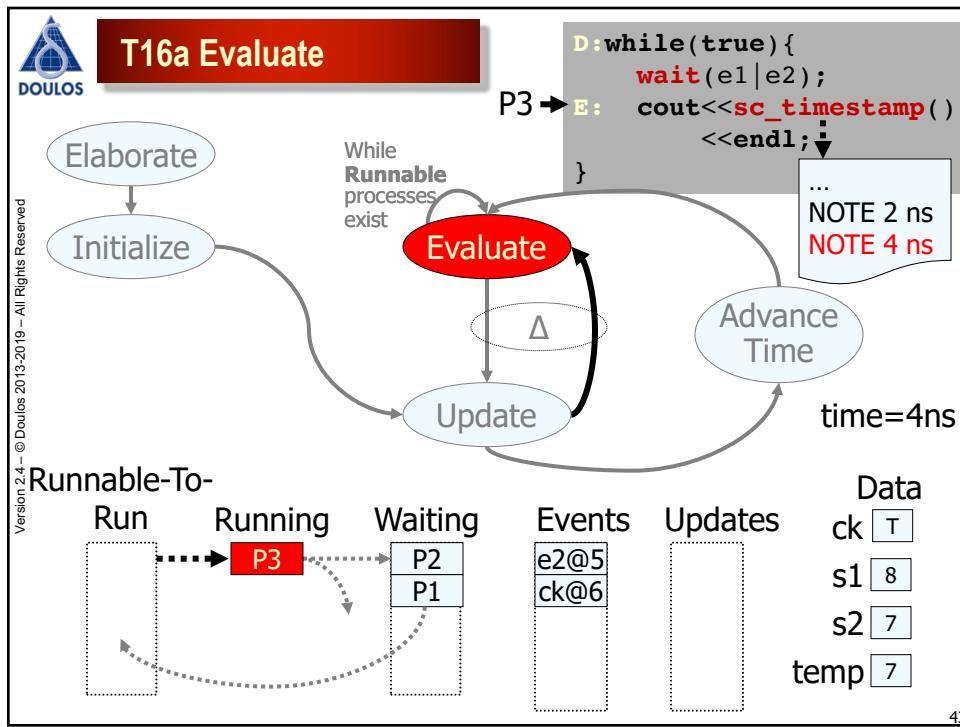


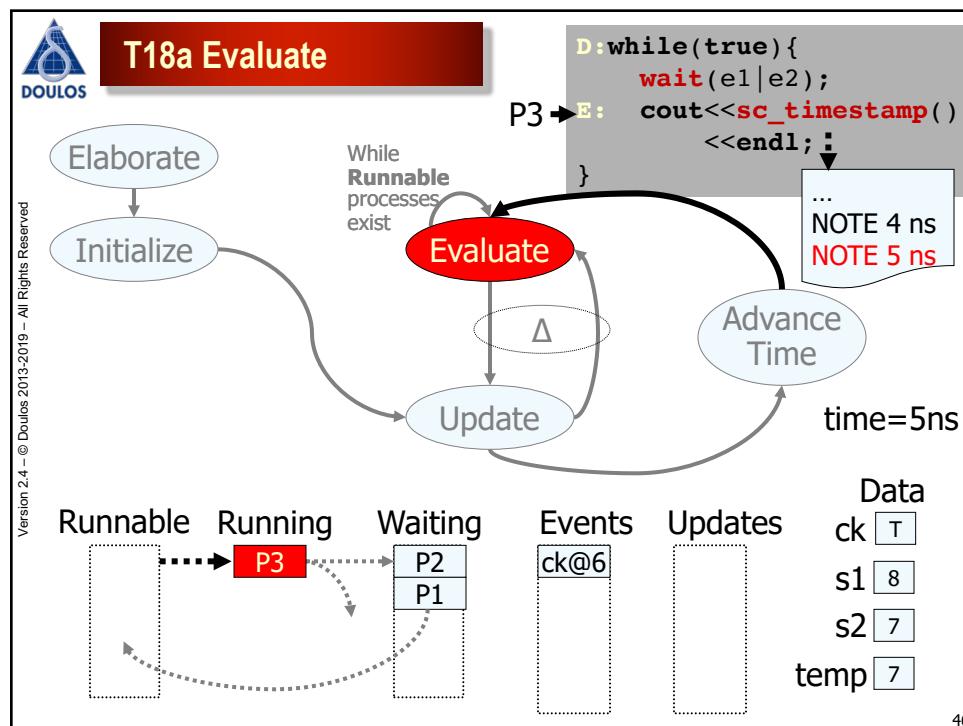
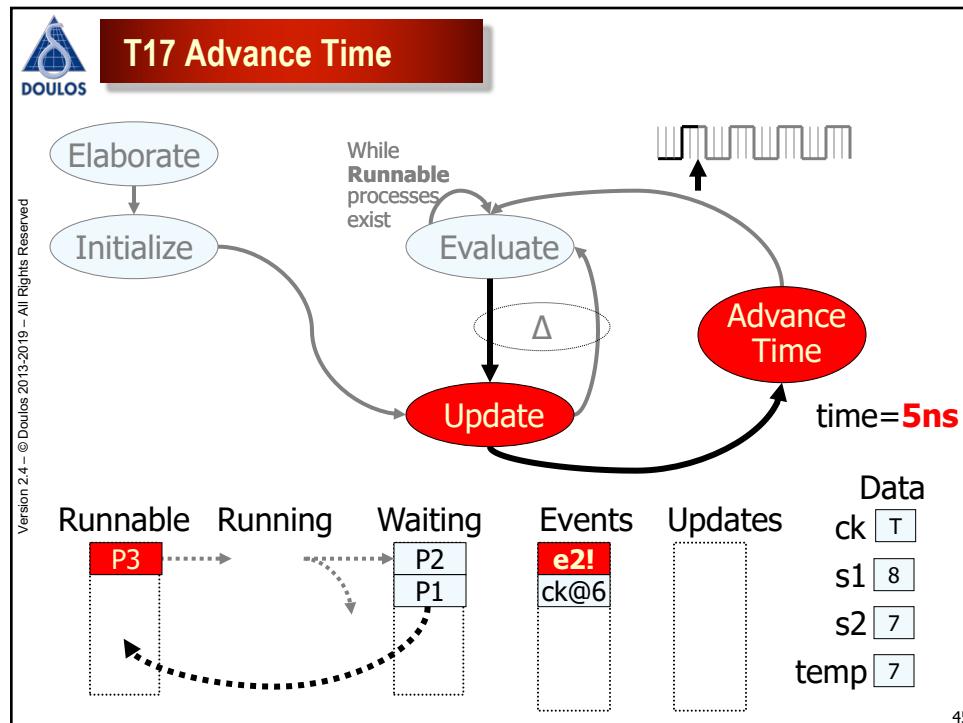


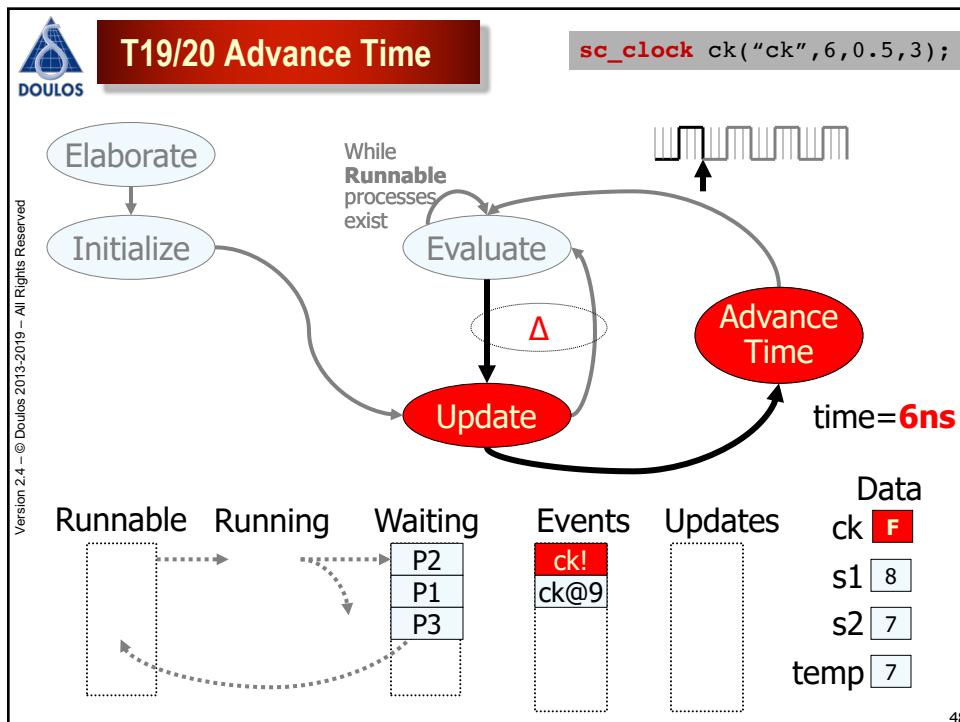
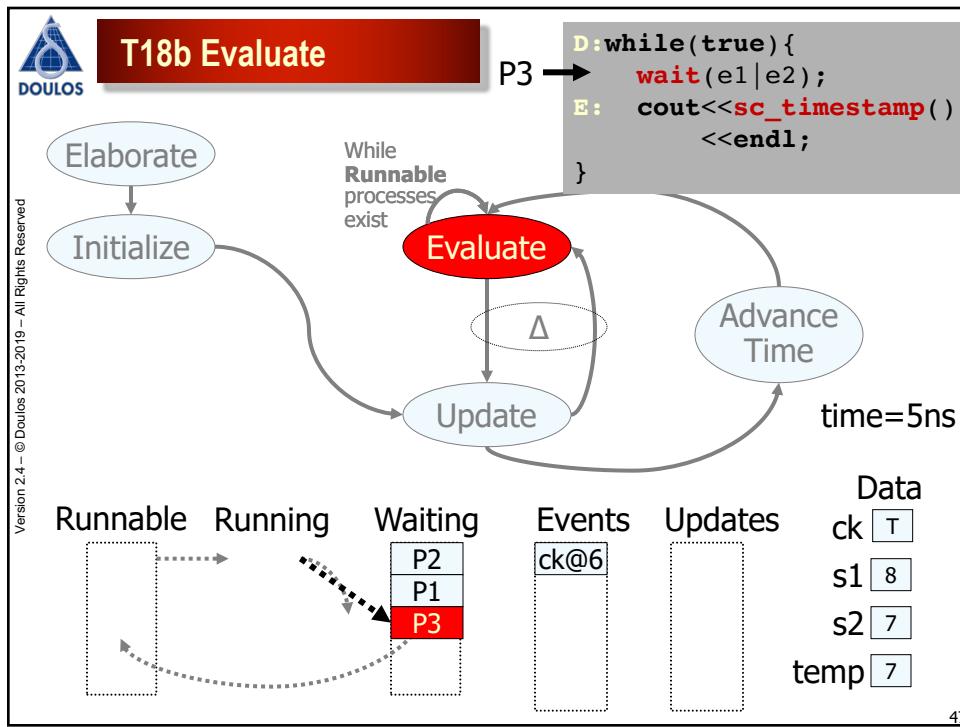


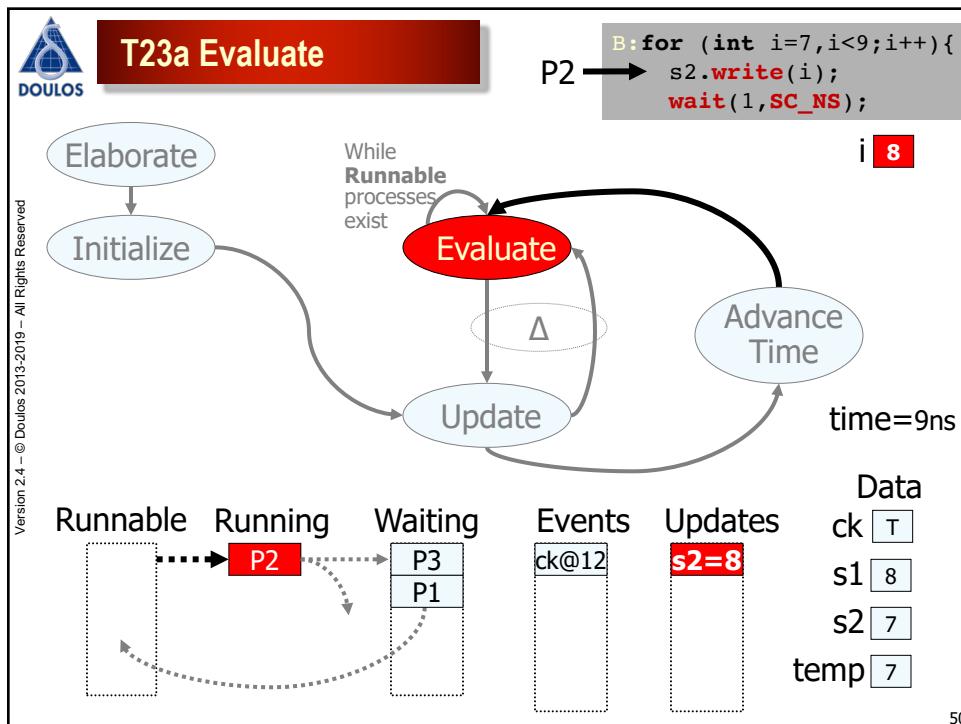
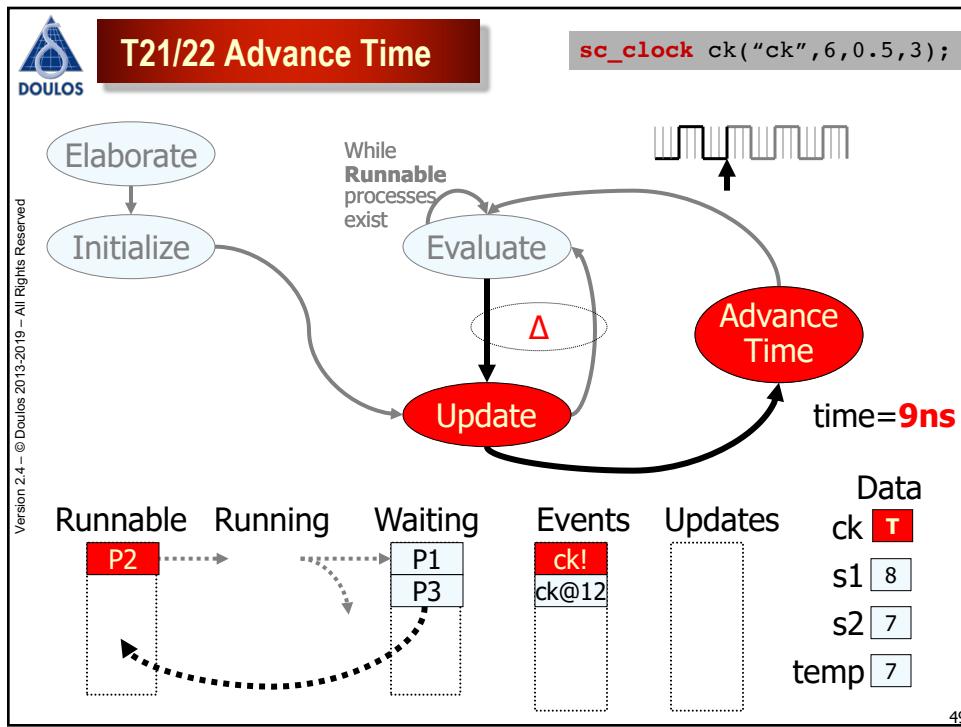












...and on and on it goes...

- Simulation returns when
 - No more **Runnable** nor **Waiting** processes
 - Encounters **sc_stop()** or **SC_REPORT_FATAL(...)**
 - Simulated **sc_pause()** or time exceeds specified **sc_start(tMAX)** time
 - Simulated time exceeds 64 bits
 - 2^{64} ps = 30 weeks 3 days 12 hours 5 min 44 sec
 - Really bad stuff
 - Explicit **exit()** or **abort()**
 - unhandled exception (control-C) or **kill**
 - bad memory access (pointer gone wrong)
 - out of memory (leak)
 - stack overflow

51

Observations

- Sample code does not follow best practices
 - designed to be short enough to fit on one page
 - only a single module
 - **sc_in**, **sc_out** discouraged
 - prefer **sc_port< sc_signal_inout_if< T >>**
 - unless you need the event finder specialization
 - **sc_signal** is a low-level construct (RTL)
 - **sc_clock** slows simulation - too much context switching
- Serious ESL simulations
 - involve 10's to 1,000's of processes
 - contain lots of module hierarchy and interconnect
 - use higher level of abstraction (TLM)
 - don't have explicit clocks

52

Process order of execution

- Processes model simulated concurrency (parallel execution)
- Each SystemC implementation has own initial ordering
 - important to allow reproducible results
 - should not depend on this behavior
- Can change this ordering by shuffling process registration
- Design required dependencies should be dictated by explicit events and handshakes

```

sc_CTOR(M) //force random
...
if (randomize) {
    switch (random()%3) {
        case 0: SC_THREAD(P3_thrd);
            break;
        case 1: SC_THREAD(P2_thrd);
            sensitive<<ckp.pos();
            break;
        case 2:
            SC_METHOD(P1_meth);
            sensitive<<s2;
            dont_initialize();
    } //endcase
} //endif
} //end SC_CTOR

```

53

Guaranteeing order

- Would using simple int instead of sc_signal<int> work?

```

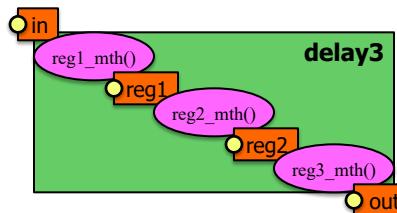
SC_MODULE(delay3) {
    sc_in<int> in_port;
    sc_out<int> out_port;
    sc_signal<int> reg1, reg2;
    void reg1_mth(void) {
        v1.write(in_port->read());
    }
    void reg2_mth(void) {
        v2.write(v1.read());
    }
    void reg3_mth(void) {
        out_port->write(v2.read());
    }
}

```

```

sc_CTOR(delay3) {
    SC_METHOD(reg2_mth);
    sensitive << in_port;
    SC_METHOD(reg1_mth);
    sensitive << in_port;
    SC_METHOD(reg3_mth);
    sensitive << in_port;
}

```



54

How sc_signal works

- During update kernel calls `update()` method on all objects that did `request_update()` in preceding delta cycle
- `write()` is equivalent to verilog non-blocking assignment

```
mysig.write(expr);
myvar <= expr;
```

```
struct buffer
: sc_prim_channel, buffer_if {
    int read(void) {return curr; }

    void write(int v) {
        next = v; request_update();
    }

    void update(void) {
        curr = next; evt.notify();
    }

    sc_event& written_evt(void) {
        return evt;
    }

private:
    int curr, next;
    sc_event evt;
};
```

55

For further study

- Download
 - Examine the basic code discussed
 - Examine the output log file
 - Examine the instrumented code used for the above
 - Includes the source to reproduce results
 - <https://github.com/dcblack/engine>
- Includes this presentation
- Try to get different results
 - Reorder the process registration code
 - Try a different platform, simulator vendor, or version



Common C++ Coding Pitfalls

- Forgetting semi-colon (;) on class/struct
- Forgetting to tag `private`, `protected` or `public`
- Copy-paste error on header guard (or completely forgetting)
 - HINT: Learn how to use `#pragma once`
- Failure to implement code separate from declaration
- Attempting construction inside the declaration
- Forgetting class-name qualifier when implementing separately
- Direct instantiation of class members leading to excess header needs
- Failure to use pass-by-reference (esp. for polymorphism)
- Abuse of pointers leading to memory faults
- Omitting the constructor or destructor
- Omitting copy-constructor or `operator=`
- Using `printf` instead of `boost::format` or `fmt`
- Failure to utilize STL or boost
- Not using `const`, and/or using `#define`

Version 2.4 – © Doulos 2013-2019 – All Rights Reserved

57



Common SystemC Coding Pitfalls

- Templating `sc_port` on non-`sc_interface` type
- Forgetting to bind (connect) all ports/sockets
- Forgetting to register processes inside constructor
- Ports use dot (.) operator instead of arrow (->) operator
- Using blocking functions inside `SC_METHOD` processes
- Incorrectly locating channels relative to `sc_port` or `sc_export`
- Coding at RTL level - too much detail
- Infinite loop path missing `wait` in `SC_THREAD` process
- Attempting to `sc_stop` and restart with `sc_start`
- Simulation phase actions during elaboration phase
- Using `std::cout` instead of `SC_REPORT_INFO`
- Using `std::cerr` instead of `SC_REPORT_ERROR`
- Elaboration phase actions during simulation phase
- Excessive context switching or I/O causing simulation to crawl
- Converting `SC_THREADS` to `SC_METHODS` to gain performance without first profiling code to determine real cause of slowdowns

Version 2.4 – © Doulos 2013-2019 – All Rights Reserved

58



SystemC Guidelines

- Abstract as high as possible – only add required details
 - Ask: What am I trying to answer with this model?
- Code as simply and cleanly as possible
 - State machines and wires are messy and hard to debug
- Avoid too much context switching (think)
- Be careful with immediate notification (delayed is safer)
- Communicate between processes with channels
- Communicate across module boundaries with ports
- Use indirect instantiation for flexibility (`std::unique_ptr<>`)
- Limit details - Model to requirements
- Improve your C++ coding skills



Questions



www.doulos.com