# Model Checking and Co-simulation of a Dynamic Task Dispatcher Circuit using CADP

**Etienne Lantreibecq**
STMicroelectronics
http://www.st.com

**Wendelin Serwe**
INRIA and LIG / VASY
http://vasy.inria.fr

# Introduction

- increasing complexity of architectures for mobile multi-media applications
- globally asynchronous, locally synchronous
- costly design errors:
  errors to be found as early as possible
- validation of complex *control* blocks:
  - formal verification not addressed by the CAD tools used by STMicroelectronics
  - current practice: simulation

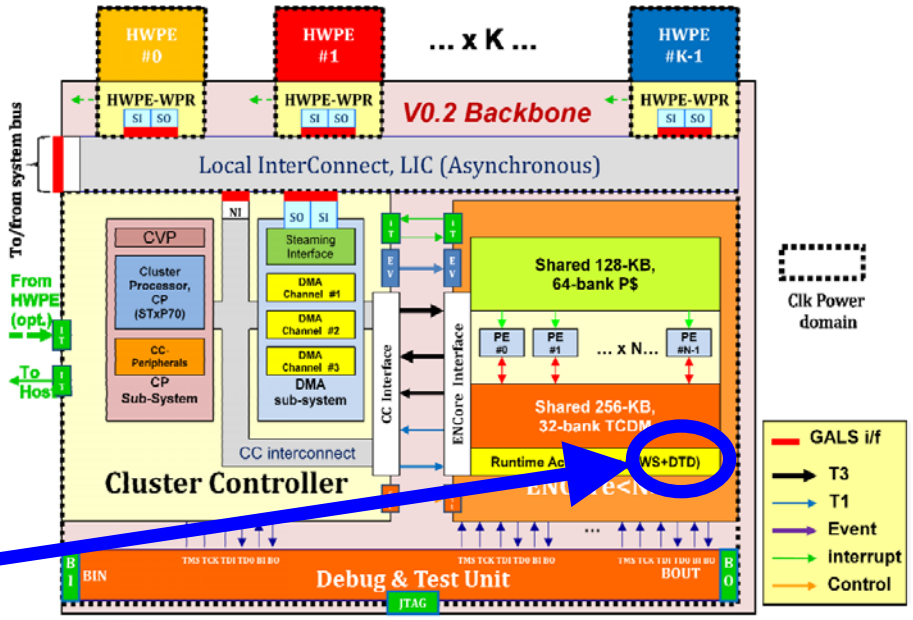
*this talk*: case-study on a hardware block designed by STMicroelectronics using CADP

# Plan of the presentation

- presentation of the dynamic task dispatcher
- formal modeling using LNT
- model checking using MCL
- co-simulation of the C++ and LNT models
- conclusion

# Context: “Platform 2012” project

- many-core programmable accelerator
- 16-processor cluster for fine-grain parallelism (shared data memory and instruction cache)
- tasks divided in parallel executable sub-tasks (same code, different data) “*dup (f, i)*”: *i* instances of function *f*
- task programming model “ready to run until completion”:
  - no sub-tasks interaction
  - any ordering of sub-tasks
- dedicated hardware to switch tasks in only few clock cycles: DTD

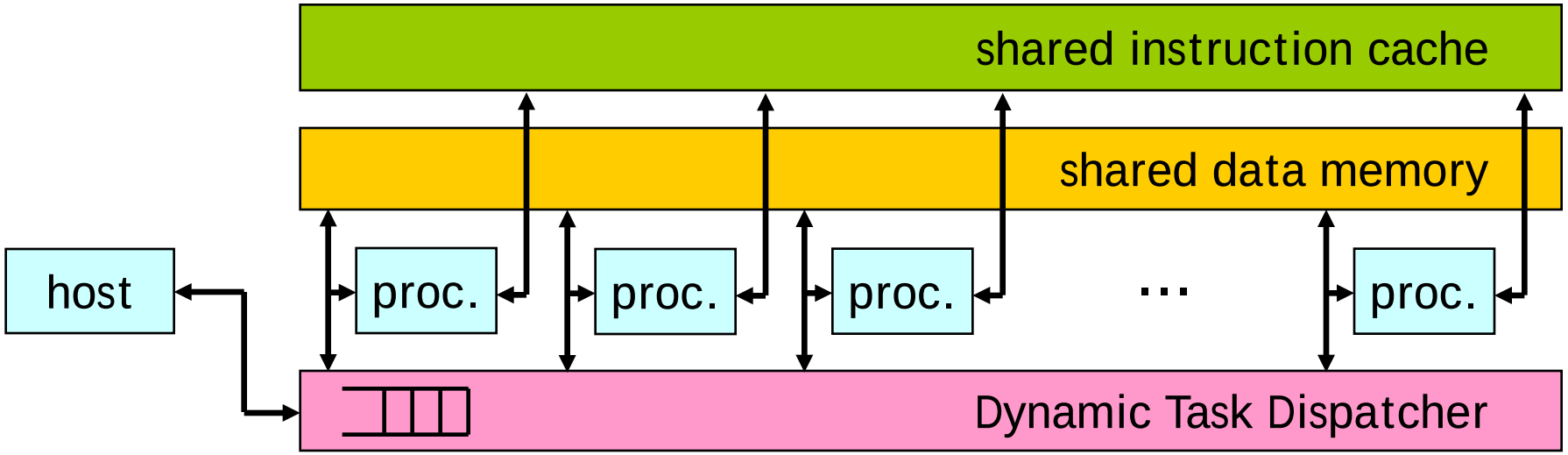



http://www.2parma.eu/images/stories/p2012_whitepaper.pdf

# Dynamic Task Dispatcher (DTD)

- dispatch tasks on idle processors
  - queue for task-request of the host
  - sub-tasks requested by processors of the cluster (at most three levels of sub-tasks)
- wake-up processors as needed
- processor-DTD communication using standard load/store on dedicated addresses

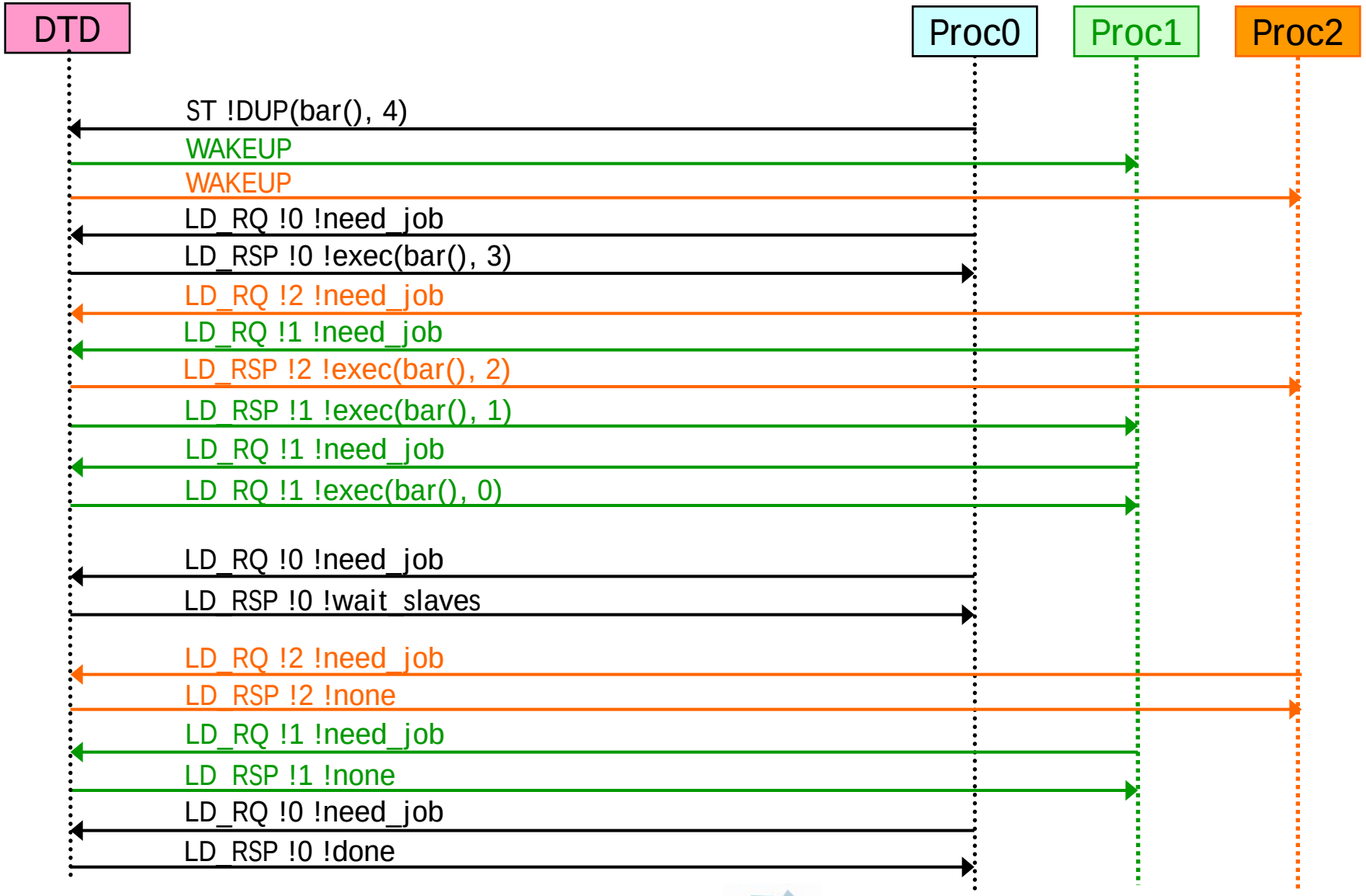
thanks to Michel Favre for discussion on the DTD



# DTD: interactions with a processor

- *store*:
  - ST (*dup* (*pc*, *i*)): request to execute *pc* *i*-times
  - ST (*boot*): a processor signals is ready to execute
- *load*: two phases (request – response)
  - LD_RQ (*need_job*): request a task
  - LD_RSP (*exec* (*pc*, *i*)): task *pc* with index *i*
  - LD_RSP (*none*): no more work left (go to sleep)
  - LD_RSP (*wait_slave*): wait for sub-tasks
  - LD_RSP (*done*): all sub-tasks finished
- *wakeup* (WAKEUP): activate the processor

# Execution scenario



# Formal Modeling using LNT

# LNT (LOTOS NT) language

- integration of the features of
  - process algebras
  - imperative programming languages
- easy-to-learn, user-friendly syntax
- formal semantics
- recommended input language for CADP (lnt.open)
  - compilation to LOTOS & EXEC/CÆSAR
  - generation of the labeled transition system
  - connection to on-the-fly verification tools
- reference manual:
  ftp://ftp.inrialpes.fr/pub/vasy/publications/cadp/Champelovier-Clerc-Garavel-et-al-10.pdf

# Modeling approach

- scenarios to avoid state space explosion
  - represent constraints on applications
  - abstract processors
  - abstract data & memory
- scenario description as LNT types & functions
- both hardware and software as LNT processes
- no global clock

# Representing simultaneous signals

- DTD clock based:
  several signals per clock cycle possible
- no immediate reaction on a single event
- multi-phase approach:
  - accept inputs and memorize their presence
  - take decisions and/or compute outputs:
    internal transition on a particular gate
  - asynchronously propose outputs
- representation of all possible interleavings

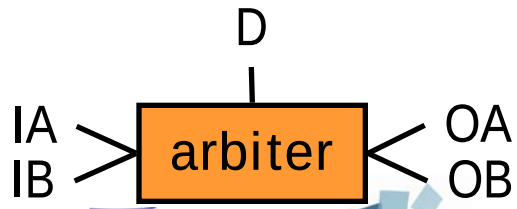
# Modeling style: example

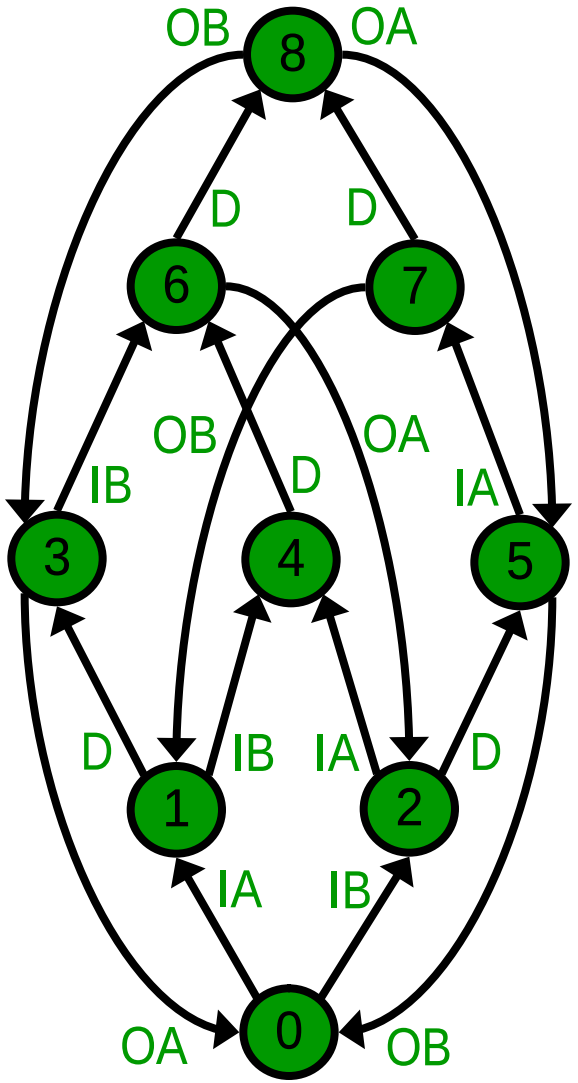
```
process Arbiter [IA, OA, IB, OB, D: none] is
  var sa, sb: Nat in
    sa := 0; sb := 0;
    loop select
    (* handling inputs *)
       when sa == 0 then IA; sa := 1 end when
    [] when sb == 0 then IB; sb := 1 end when
    (* decision *)
    [] when sa == 1 then sa := 2 else
       when sb == 1 then sb := 2 end when ;
       D
    (* handling outputs *)
    [] when sa == 2 then OA; sa := 0 end when
    [] when sb == 2 then OB; sb := 0 end when
    end select end loop
  end var
end process
```

no input

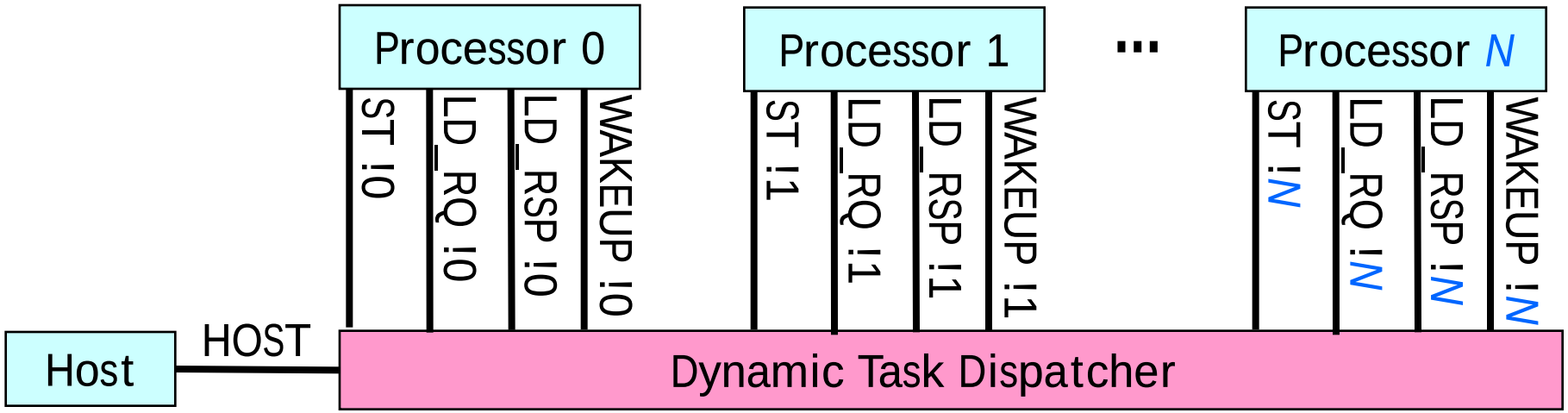
input arrived

input accepted



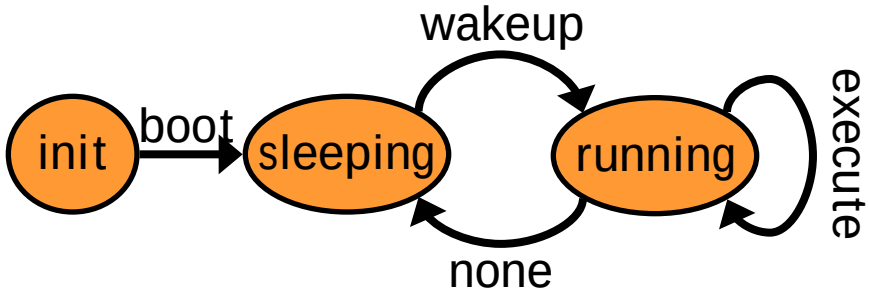



# Architecture of the LNT model




Processors described by three LNT processes:

- Processor: status automaton




- Dup: implementation of *dup*()
- Execute: scenario-specific definition of instructions

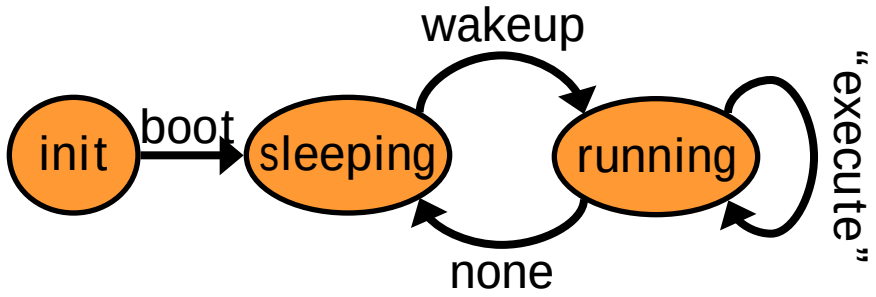
# Scenario description in LNT

```
type PC is pc_1, pc_2, pc_3 with "==", "!=" end type

process Execute [ST, LD_RQ, LD_RSP: any]
                (j:Job, inout s:Job_Stack)
is
  var pc: PC in
    pc := get_PC (j);
    case pc in
      pc_1 -> Dup [ST, LD_RQ, LD_RSP]
                  (pc_3, 4, dup (pc_2, -1), !?s)
    |pc_2 -> (* instructions of the duplicated function *)
    |pc_3 -> (* instructions of the continuation *)
    end case
  end var
end process
```

mapping between program counters and instructions to execute

# LNT processes Dup and Processor

```
process Dup [ST, LD_RQ, LD_RSP: any]
                (pc: PC, i: Int, c: Job, inout s: Job_Stack) is
  s := push (c, s); ST (dup (pc, i))
end process

process Processor [ST, LD_RQ, LD_RSP, WAKEUP: any] is
  var s: Job_Stack := {} in ST (boot); loop
    WAKEUP;
    loop l in var j: Job in
      LD_RQ (need_job); LD_RSP (?j);
      case j in var pc: PC, i: Int in
        exec (pc, i) ->
      |wait_slave -> null
      |done -> if empty(s) then break l else
                  j := head(s); s := pop(s); Execute [ST, LD_RQ, LD_RSP] (j, !?s)
               end if
      |none -> break l
      end case
    end var end loop
end loop end var end process
```



# Modeling approach

- hand-written model for four processors
- development of a model generator
  - parameterized by the number of processes
  - generation of models for 4, 6, and 16 processors
- different versions
  - fit to CADP tools
  - experiment complex optimizations

# State space generation: scenarios

- *N* = number of available processors
  total number of tasks > *N* for all scenarios
- Scenario 1: without *dup*()
- Scenario 2: one task forking subtasks
  - variant 2_1: more subtasks
  - variant 2_2: more tasks without *dup*()
- Scenario 3: nested calls to *dup*()
  - variants 3_1 & 3_2: different number of subtasks per level of nested *dup*()
  - variant 3_3: more main tasks without *dup*()
- Scenario 4: consecutive calls to *dup*()
  - variant 4_1: more subtasks at each invocation
- Scenario 5: two main tasks calling *dup*()

| *N* | scenario | states | transitions |
|---|---|---|---|
| 4 | 1 | 664,555 | 2,527,653 |
| | 2 | 28,032 | 91,623 |
| | 2_1 | 73,984 | 255,391 |
| | 2_2 | 920,649 | 3,537,763 |
| | 3 | 168,466 | 557,363 |
| | 3_1 | 1,445,922 | 5,204,671 |
| | 3_2 | 655,546 | 2,387,195 |
| | 3_3 | 4,435,309 | 17,328,979 |
| | 4 | 63,760 | 211,579 |
| | 4_1 | 168,288 | 586,539 |
| | 5 | 181,170 | 596,022 |
| | 5_1 | 1,626,933 | 5,989,205 |
| 6 | 2 | 4,998,344 | 24,324,439 |
| | 2_1 | 14,778,488 | 74,826,343 |
| | 4 | 12,696,086 | 62,482,651 |
| | 4_1 | 37,090,190 | 189,595,795 |
| | 5 | 97,297,953 | 489,846,494 |

# Model Checking using MCL

# Model checking: property 1

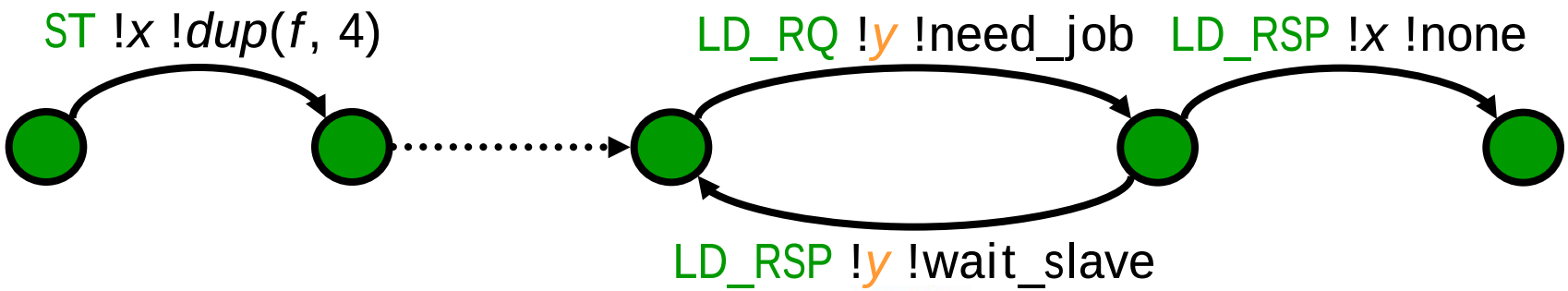
- the scenario terminates:
  $\mu X$ . [**true**] $X$

  *false* for all scenarios with a Dup operation

thanks to Radu Mateescu for help with MCL

- the scenario terminates, ignoring "active waiting":
  $\mu X$ . ( [**true**] $X$ **or** ( **exists** $y$:Nat . **< true*** . {LD_RSP !$y$ !"wait_slave"} **>@** ) )

  *true* for all scenarios



# Further verified MCL properties

- property 2: for each processor woken up, eventually there is no more work left

  [**true*** . {WAKEUP ?*x*:Nat}]
  inevitable ({LD_RSP !*x* !"none"})

  definition of inevitable in the paper

- property 3: each call to *dup*() executes to completion

  [**true*** . {ST ?*x*:Nat !"dup"}]
  inevitable ({LD_RSP !*x* !"done"})

- property 4: each task of the host is executed exactly once

  [**true***. {HOST ?*c*:String}]
  ( inevitable ({LD_RSP ?*x*:Nat !*c*}) **and**
  [(**true*** . {LD_RSP ?*y*:Nat !*c*){2}] **false** )

# Co-simulation

# Co-simulation goals

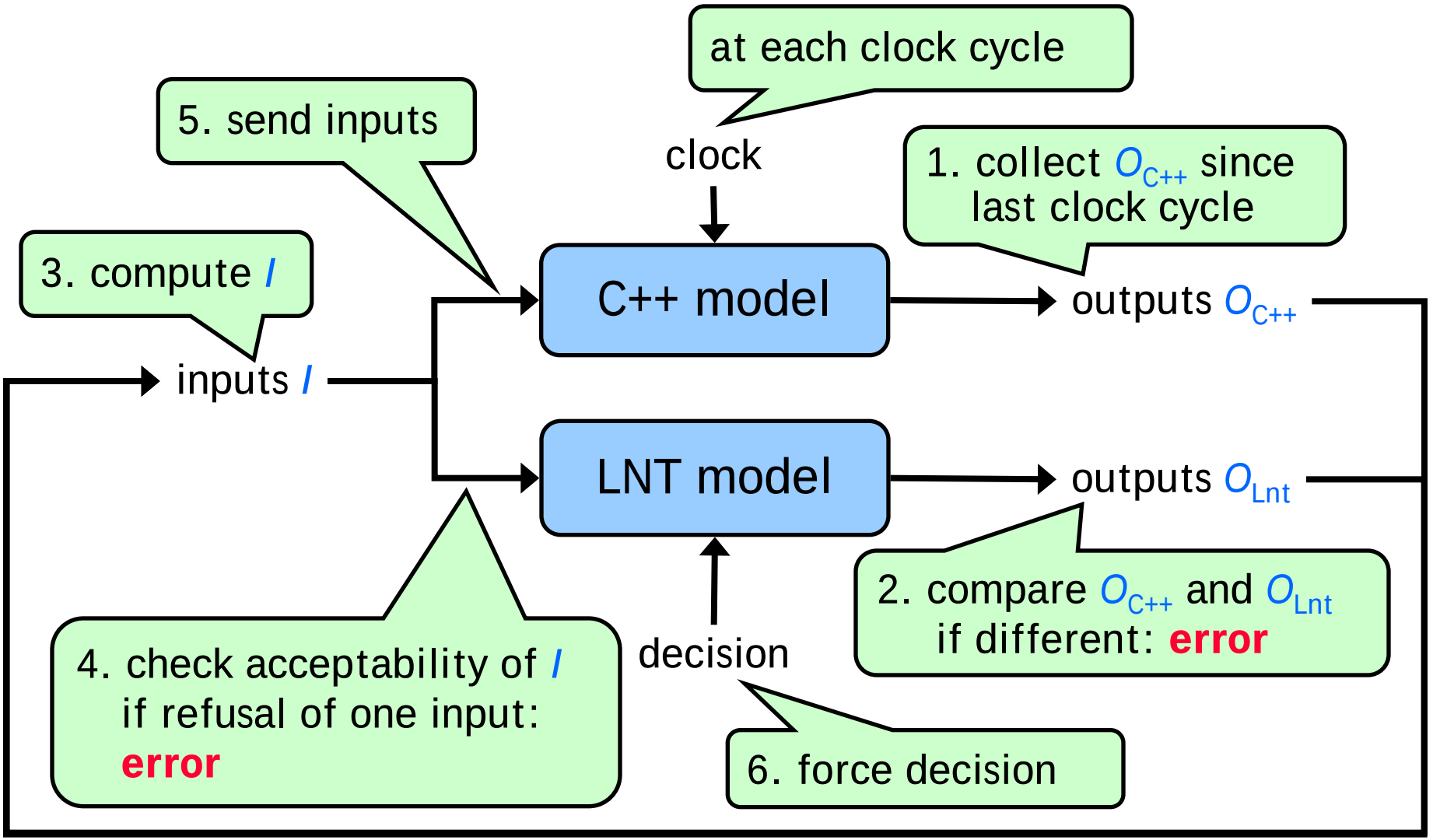
- simultaneous execution / mutual cross-checking of:
  - the architect's design:
    synthesizable C++ code
  - the formal, verified LNT model:
    C code generated using the EXEC/CÆSAR framework
- reuse the architect's simulation environment
- main challenges:
  - *connection between synchronous and asynchronous*
  - arbitration decisions taken on clock signals
  - varying number of signals per clock cycle
  - choose one interleaving of the signals

# EXEC/CÆSAR framework

- translation of a LNT model to a C function *f*()
- rendezvous = call to a Boolean gate function
  - gate function parameters to exchange values
  - rendezvous accepted iff gate function returns true
- given a state *s*, *f*():
  - computes the set of outgoing transitions *O*
  - signals a deadlock if *O* = {}
  - iterates over the elements of *O*, calling gate functions
  - moves to next state when a rendezvous is accepted
  - allows to start over if no rendezvous is accepted

enables to compute the set of proposed rendezvous

# Co-simulation scheme



# Conclusion

- case-study of a complex industrial hardware block
- formal LNT model
  - **developed by engineers - understandable by designers**
  - discuss problems and experiment optimizations
  - increase confidence in both models (LNT & C++)
- model checking
  - express complex properties
  - verify all interleavings (instead of testing only some)
- co-simulation
  - mutual cross-check of both models (LNT & C++)
  - uncovered ambiguity in natural language specification

LNT: formal modeling of complex control blocks practically feasible!

# Thank you !

for more information about CADP, LNT, and MCL

- http://vasy.inria.fr/cadp
- http://cadp.forumotion.com