Guidelines for Producing Concise LNT Models, Illustrated with Formal Models of the Algorand Consensus Protocol

Hubert Garavel

Inria Grenoble – LIG

http://convecs.inria.fr

informatics mathematics

inria

LIG

Outline

- 1. INTRODUCTION TO ALGORAND
- 2. FORMAL MODELS OF THE ALGORAND CONSENSUS PROTOCOL
  - The LNT language
  - History of versions for the formal models
  - From version U0 to version U1
  - From version U1 to version U2
  - From version U2 to version U3
  - From version U3 to version U4
- 3. VERIFICATION BY STATE-SPACE EXPLORATION
  - Visual checking
  - Equivalence checking
  - Model checking
- 4. CONCLUSION

1. INTRODUCTION TO ALGORAND

Blockchain challenges

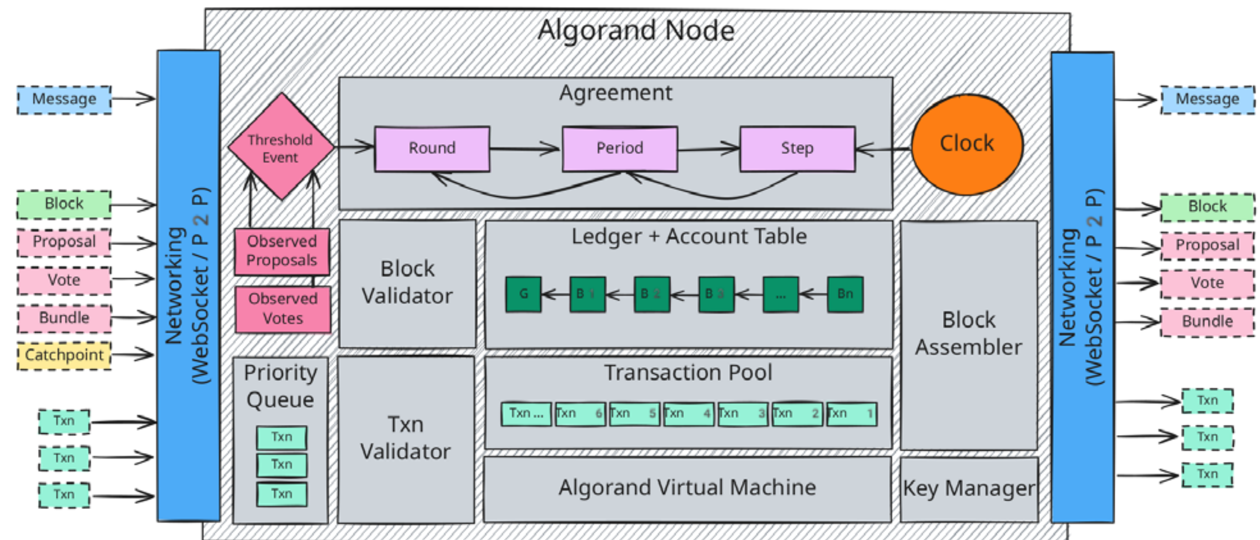
- Quadrilemma problem:
  - decentralization
  - scalability
  - security
  - low energy consumption

- Different proposals:
  - Bitcoin
  - Ethereum, Solana, Algorand, Cardano, etc.

Algorand ecosystem

- Algorand means different things:
  - a secure distributed ledger algorithm
  - a high-performance Layer 1 blockchain
  - a company : https://algorand.co
  - two cryptocurrencies: ALGO (€950M), EURD (€2.6M)

The AlgoKit app



Algorand consensus protocol

- Consensus among distributed processes
  - proof-of-stake: committees play a crucial role
- Original version: BBA* (general Binary Byzantine Agreement)
  - defined in [Chen & Micali 2019]

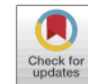
Theoretical Computer Science 777 (2019) 155–183

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Algorand: A secure and efficient distributed ledger ☆



Jing Chen [a,*], Silvio Micali [b]

[a] Computer Science Department, Stony Brook University, Stony Brook, NY 11794, USA
[b] CSAIL, MIT, Cambridge, MA 02139, USA

Excerpt from [Chen & Micali 2019]

Step s, $5 \le s \le m$, $s-2 \equiv 0 \mod 3$: A Coin-Fixed-To-0 Step of $BBA^\star$

For every user $i \in PK^{r-k}$: i starts his own Step s as soon as he finishes his own Step $s-1$.

- User i waits a maximum amount of time 2λ.[a] While waiting, i acts as follows.
  - *Ending Condition 0:* If at any point there exists a string $v \neq \perp$ and a step s' such that
    - (a) $5 \le s' \le s$, $s'-2 \equiv 0 \mod 3$—that is, Step s' is a Coin-Fixed-To-0 step,
    - (b) i has received $\ge t_H$ valid messages $m_j^{r,s'-1} = (ESIG_j(0), ESIG_j(v), \sigma_j^{r,s'-1})$, and
    - (c) i has received a valid message $(Head(B_\ell^r), \sigma_\ell^{r,1})$ where ℓ is the *second* component of v,

    then, i stops waiting and ends his own execution of round r right away; sets $H(B^r)$ to be the *first* component of v; and sets his own $CERT^r$ to be the set of messages $m_j^{r,s'-1}$ of sub-step (b) together with $(Head(B_\ell^r), \sigma_\ell^{r,1})$.[b]
  - *Ending Condition 1:* If at any point there exists a step s' such that
    - (a') $6 \le s' \le s$, $s'-2 \equiv 1 \mod 3$—that is, Step s' is a Coin-Fixed-To-1 step, and
    - (b') i has received $\ge t_H$ valid messages $m_j^{r,s'-1} = (ESIG_j(1), ESIG_j(v_j), \sigma_j^{r,s'-1})$,[c]

    then, i stops waiting and ends his own execution of round r right away without propagating anything as a (r,s)-verifier; sets $B^r = B_\epsilon^r$; and sets his own $CERT^r$ to be the set of messages $m_j^{r,s'-1}$ of sub-step (b') together with $Head(B_\epsilon^r)$.
  - If at any point he has received $\ge t_H$ valid $(r, s-1)$-messages $m_j^{r,s-1}$ of the form $(ESIG_j(1), ESIG_j(v_j), \sigma_j^{r,s-1})$, then he stops waiting and sets $b_i \triangleq 1$.
  - Otherwise, when time 2λ runs out, i sets $b_i \triangleq 0$.[d]
  - Once the value b_i is set, i computes Q^{r-1} from $CERT^{r-1}$ and checks whether $i \in SV^{r,s}$.
  - If $i \in SV^{r,s}$, i computes the message $m_i^{r,s} \triangleq (ESIG_i(b_i), ESIG_i(v_i), \sigma_i^{r,s})$ with v_i being the value he has computed in Step 4, destroys his ephemeral secret key $sk_i^{r,s}$, and then propagates $m_i^{r,s}$. Otherwise, i stops without propagating anything.

2. FORMAL MODELS OF THE ALGORAND CONSENSUS PROTOCOL

The LNT language

LNT

- Developed at INRIA Grenoble since 1996
- A pragmatic combination of proven concepts from
  - functional programming
  - imperative programming
  - process calculi: CSP, Occam, LOTOS, E-LOTOS...
- LNT features:
  - gentle learning curve (80% of LNT looks "normal")
  - executable, formal semantics
  - static and dynamic analyses

LNT for sequential programs

- Key features:
  - Ada-like imperative syntax (structured programming)
  - first-order functional semantics (no side effects)
  - strong typing
  - data-flow analysis (no uninitialized variables)
  - constructor types, pattern-matching, exceptions
  - custom types: arrays, ranges, lists, sorted lists, sets...
  - modules and libraries ("programming in the large")

LNT for concurrent programs

- LNT has a parallel part:
  - sequential part $\subseteq$ parallel part
  - combination of algorithmic languages + process calculi
  - compatible with bisimulation (compositional verif.)
- Key features:
  - message-passing concurrency
  - non-deterministic choice
  - parallel composition
  - n-party synchronization with multiway data exchanges
  - modelling of disruption

Two main applications of LNT

- Compiler construction
  - used in Grenoble to develop a dozen compilers
  - LNT compilers written themselves in LNT:

TRAIAN (80,000+ lines of code): front-end + LNT→C translator

LNT2LOTOS (40,000+ lines of code): LNT→LOTOS translator

- Modelling and verification of concurrent systems
  - Communication protocols
  - Distributed systems
  - Hardware circuits

LNT is supported by the CADP tools https://cadp.inria.fr

History of versions for the formal models

Formal modelling of BBA*

- [Bernardo, Esposito & al. 2025] (arXiv 2508:19452)
  - process-algebraic model of BBA*
  - with attacker model (honest vs malicious nodes)
  - with some abstractions
  - written in the CCS style
    + CSP-like parallel composition
    + probabilistic choices

- + executable version (written in LNT) of this model
  - LNT supports both CCS and CSP styles

Bernardo & Esposito model (1/2)

$$Algorand \triangleq \mathop{\|_S}_{i=1}^{n} Node_i$$

$$\text{where } S = \{receive_block_proposal, sync\} \cup \{propagate_{i,0}, propagate_{i,1} \mid 1 \leq i \leq n\} \cup \{commit_proposed_block, commit_empty_block\}$$

$$Node_i \triangleq N_i \|_{S'} C_{i,0,0}$$

$$\text{where } S' = \{propagate_{i,0}, propagate_{i,1}, ask_0, ask_1\} \cup \{reply_j \mid 0 \leq j \leq n\} \cup \{self_verify\}$$

$$\begin{aligned} C_{i,k_0,k_1} \triangleq & \sum_{j \in \{1,\ldots,n\}} propagate_{j,0} \,.\, C_{i,k_0+1,k_1} \\ & + \sum_{j \in \{1,\ldots,n\}} propagate_{j,1} \,.\, C_{i,k_0,k_1+1} \\ & + ask_0 \,.\, reply_{k_0} \,.\, C_{i,k_0,k_1} \\ & + ask_1 \,.\, reply_{k_1} \,.\, C_{i,k_0,k_1} \\ & + self_verify \,.\, C_{i,0,0} \end{aligned}$$

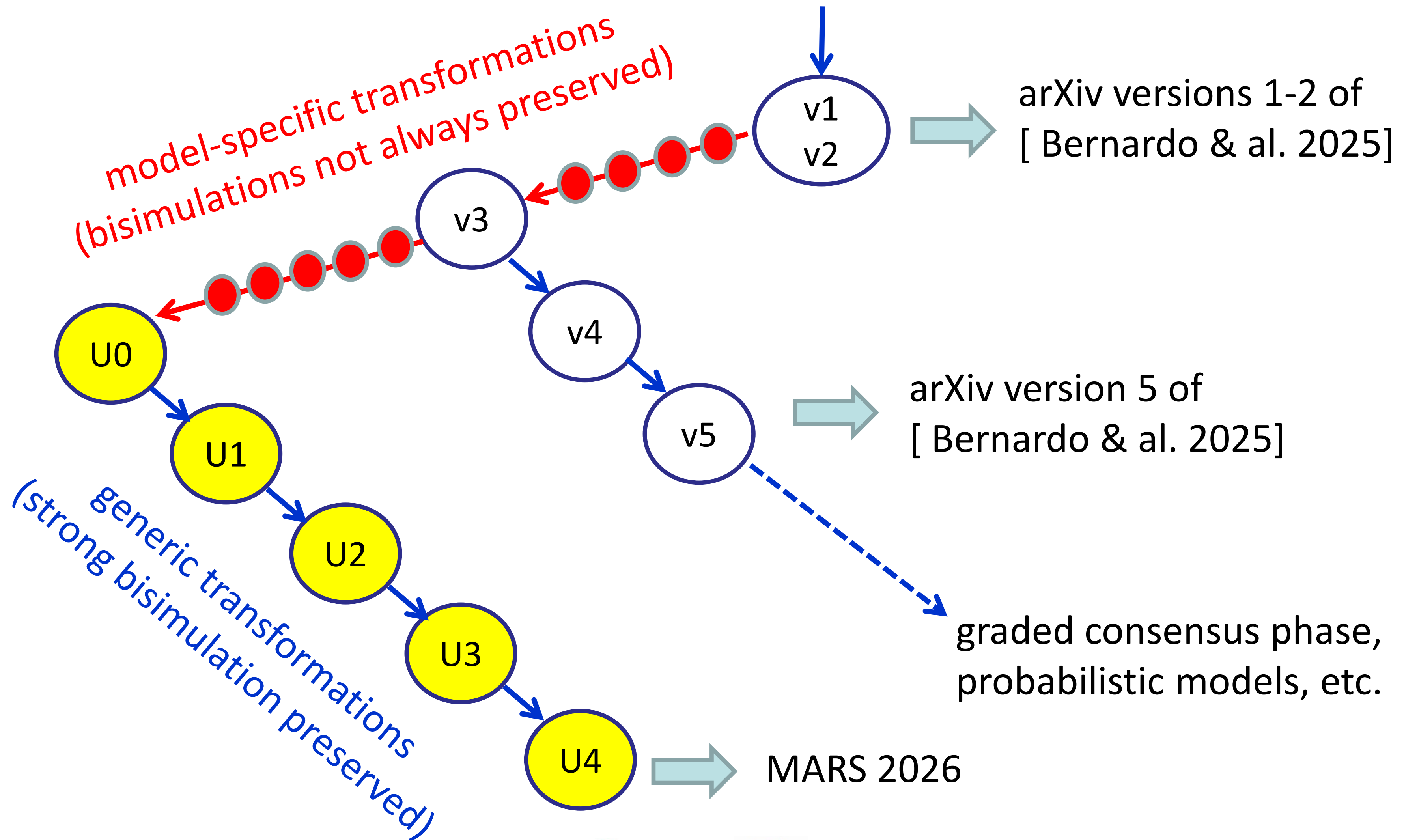
Bernardo & Esposito model (2/2)

$$
\begin{aligned}
N'_i &\triangleq compute_bit\,.\,([p_0]N''_{i,\mathbf{0}} \oplus [p_1]N''_{i,\mathbf{1}})\\
N''_{i,b} &\triangleq self_verify\,.\,sync\,.\,([p_{in}]propagate_{i,b}\,.\,sync\,.\,N'''_{i,\equiv 0} \oplus [p_{out}]sync\,.\,N'''_{i,\equiv 0})\\
N'''_{i,\equiv 0} &\triangleq ask_0\,.\,\sum_{k=0}^{n} reply_k\,.\,(k \geq t \rightarrow commit_proposed_block\,.\,N_i +\\
&\qquad\qquad k < t \rightarrow N''''_{i,\equiv 0})\\
N''''_{i,\equiv 0} &\triangleq adjust_bit\,.\,ask_1\,.\,\sum_{k=0}^{n} reply_k\,.\,(k \geq t \rightarrow N''_{i,\equiv 0,\mathbf{1}} +\\
&\qquad\qquad k < t \rightarrow N''_{i,\equiv 0,\mathbf{0}})\\
N''_{i,\equiv 0,b} &\triangleq self_verify\,.\,sync\,.\,([p_{in}]propagate_{i,b}\,.\,sync\,.\,N'''_{i,\equiv 1} \oplus [p_{out}]sync\,.\,N'''_{i,\equiv 1})\\
N'''_{i,\equiv 1} &\triangleq ask_1\,.\,\sum_{k=0}^{n} reply_k\,.\,(k \geq t \rightarrow commit_empty_block\,.\,N_i +\\
&\qquad\qquad k < t \rightarrow N''''_{i,\equiv 1})\\
N''''_{i,\equiv 1} &\triangleq adjust_bit\,.\,ask_0\,.\,\sum_{k=0}^{n} reply_k\,.\,(k \geq t \rightarrow N''_{i,\equiv 1,\mathbf{0}} +\\
&\qquad\qquad k < t \rightarrow N''_{i,\equiv 1,\mathbf{1}})\\
N''_{i,\equiv 1,b} &\triangleq self_verify\,.\,sync\,.\,([p_{in}]propagate_{i,b}\,.\,sync\,.\,N''''_{i,\equiv 2} \oplus [p_{out}]sync\,.\,N''''_{i,\equiv 2})\\
N''''_{i,\equiv 2} &\triangleq adjust_bit\,.\,(ask_0\,.\,\sum_{k=0}^{n} reply_k\,.\,(k \geq t \rightarrow N''_{i,\mathbf{0}} +\\
&\qquad\qquad k < t \rightarrow ask_1\,.\,\sum_{k=0}^{n} reply_k\,.\,(k \geq t \rightarrow N''_{i,\mathbf{1}} +\\
&\qquad\qquad\qquad k < t \rightarrow N'_i)))
\end{aligned}
$$

Limitations of the formal model

- BBA* (general Binary Byzantine Agreement) only
  - GC (Graded Consensus) phase abstracted away
  - no encryption, no ephemeral keys, no explicit VRFs
  - extra events added for observation purpose
- Timed aspects not modelled
  - synchronous assumption
  - two global synchronization barriers ("SYNC"), no timeouts
- Probabilistic aspects modelled, but not exploited
  - committee may have all sizes with the same probability
- Restricted attacker model
  - malicious nodes may only propagate wrong bit values

Many (420+) formal models of BBA*



Model-specific transformations

- 1. Presentation
  - enhance comments, spacing, order of declarations...
- 2. Simplifications
  - remove redundant "of T" annotations
- 3. Identifier renamings
- 4. Additions
  - types, constant functions, pre- and post-conditions...
- 5. Corrections
  - remove deadlocks, fix probability values
- 6. Semantic changes
  - remove events, merge two events in one, tag events with pids

Generic transformations

version	L	ℓ	$S_{1,0}$	$S_{0,1}$	$S_{4,0}$	$S_{2,2}$
U0	769	705	1740	3190	24,230	42,509
U1	607	552	1740	3190	24,230	42,509
U2	320	282	1761	3190	24,599	42,509
U3	288	250	1681	3046	20,665	34,679
U4	250	212	1723	3130	20,905	35,059
strongly minimized LTSs:			558	840	12,059	19,486

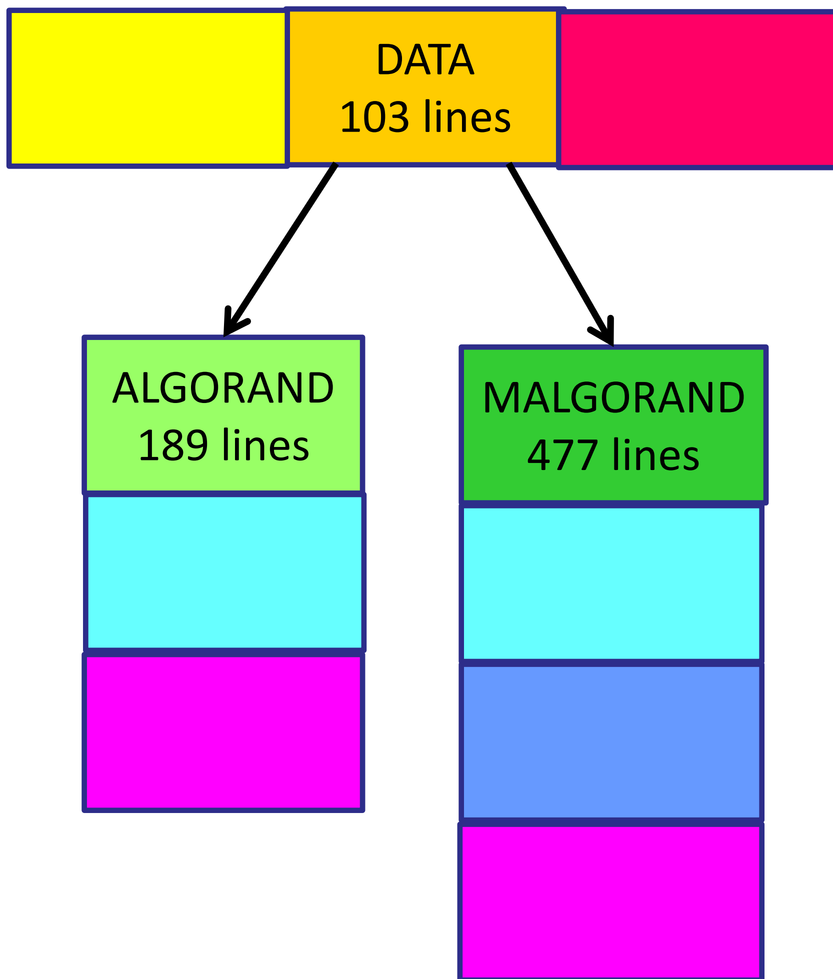
- L: number of lines ℓ: lines without blanks and comments
- $S_{1,0}$ / $S_{0,1}$: states for one honest/malicious Algorand node
- $S_{4,0}$ / $S_{2,2}$: states for 4 honest / 2 honest + 2 malicious nodes

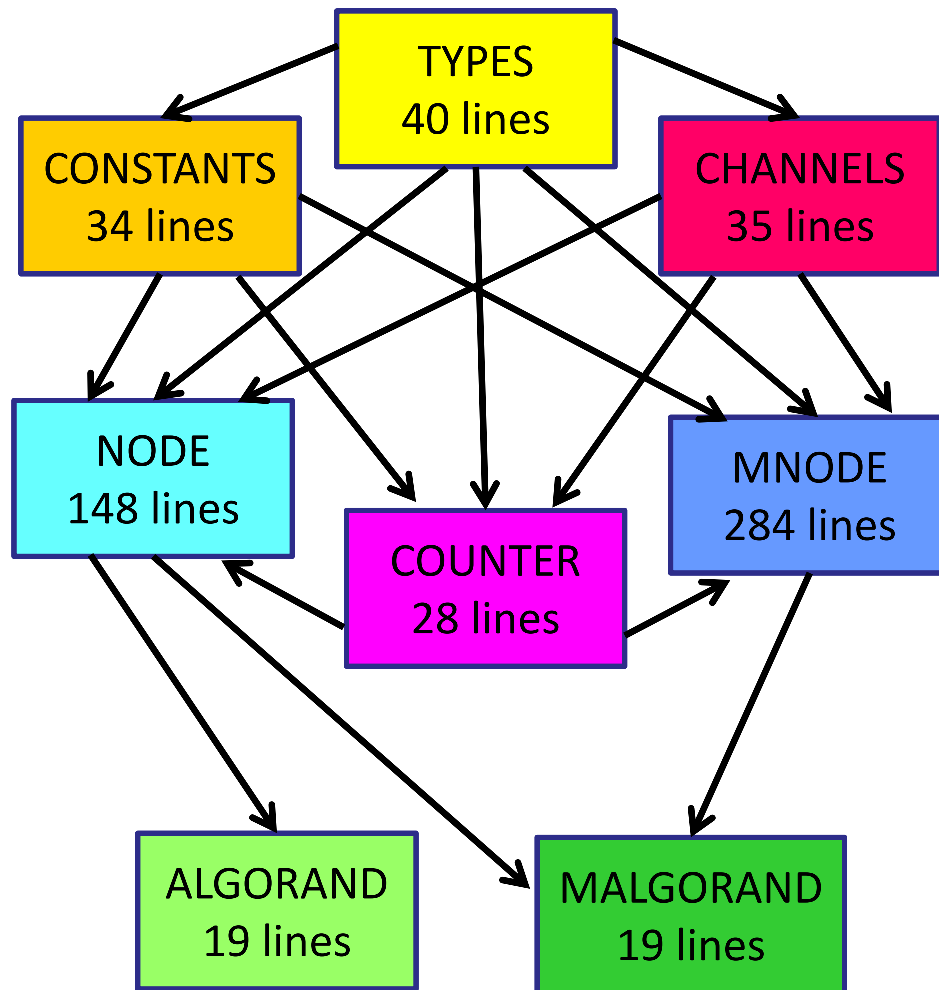
From version U0 to version U1

Modular decomposition

- Idea: factor out duplicated code using LNT modules
- Module = set of types, channels, functions, and processes
- Tree-like or dag-like dependencies between modules
  - cyclic dependencies are not permitted
  - double definitions in different modules are forbidden
  - each module must be self-contained
    (it can only use objets it defines or transitively imports)
  - "virtual" declarations help to relax the self-containment rule

U0 → U1: modular decomposition





U0: 769 lines

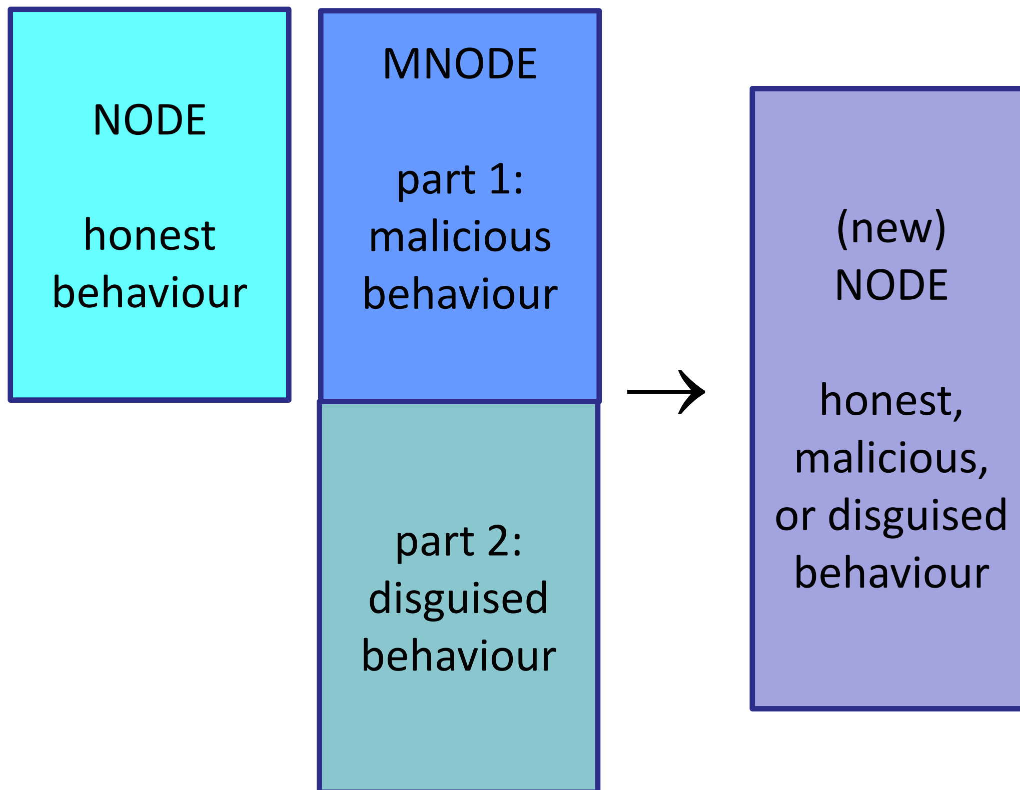
U1: 607 lines

From version U1 to version U2

Control/data tradeoff

- A state machine can be encoded in two ways:
  - "control": states are encoded using lines of code
  - "data": states are encoded using variables
- LNT example:
  - control: **alt** SEND (0) [] SEND (1) **end alt**
  - data: B := **any** BIT; SEND (B)
- Idea: introduce variables to factor out duplicated LNT code fragments

U1 → U2: control/data tradeoff



U1: 148 + 283 lines

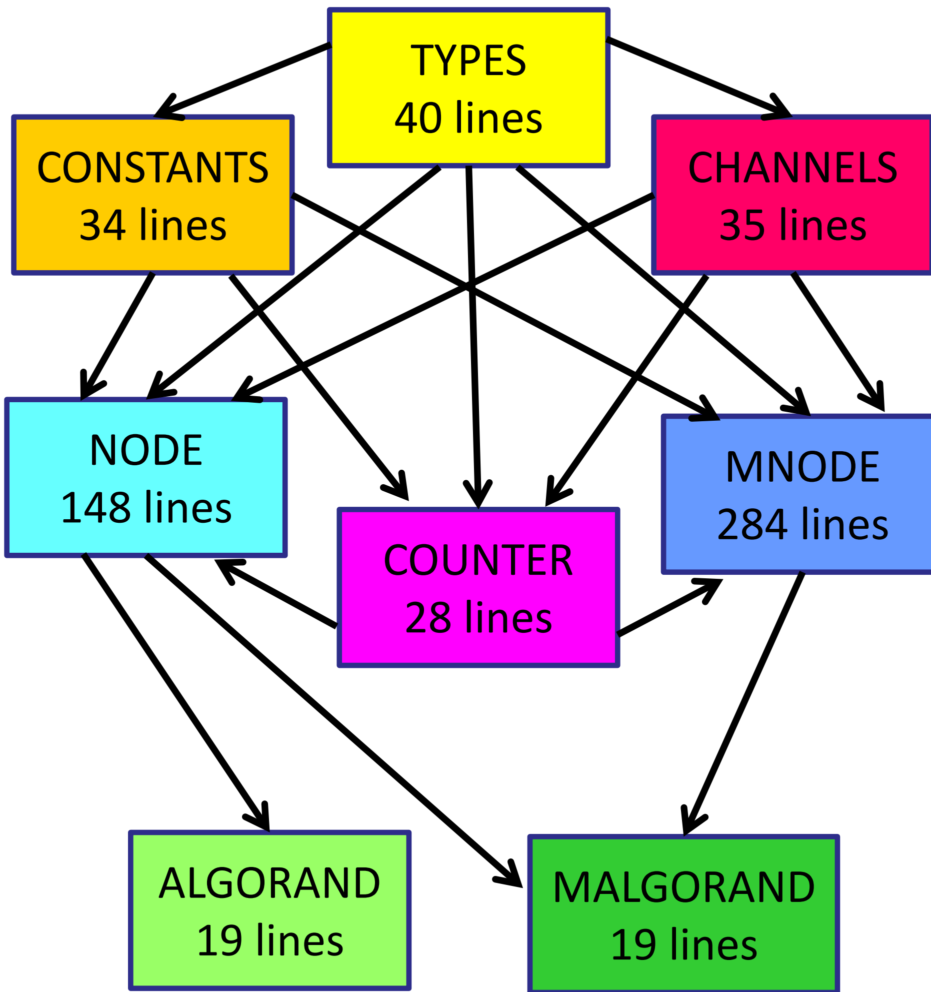
U2: 159 lines

Remove duplicated code by introducing new variables that may take 3 values:

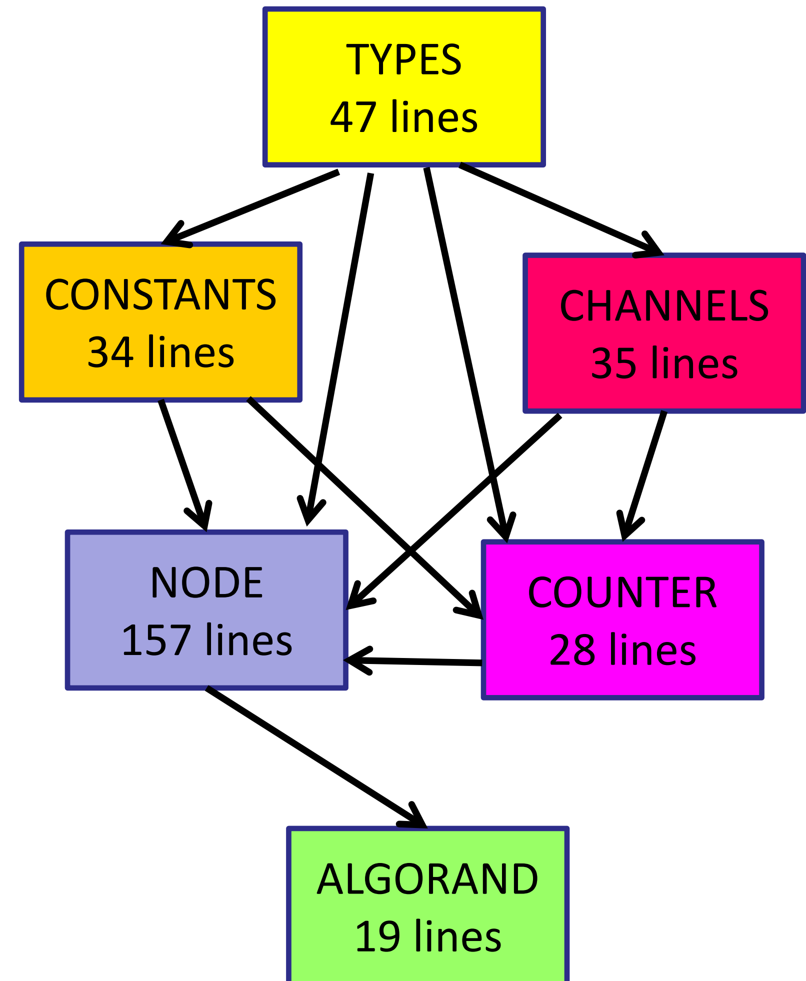
- HONEST
- MALICIOUS
- DISGUISED

Merge 3 by 3 all processes of version U1, replacing 3 similar processes by a single (parameterized) one

U1 → U2: control/data tradeoff



U1: 607 lines



U2: 320 lines

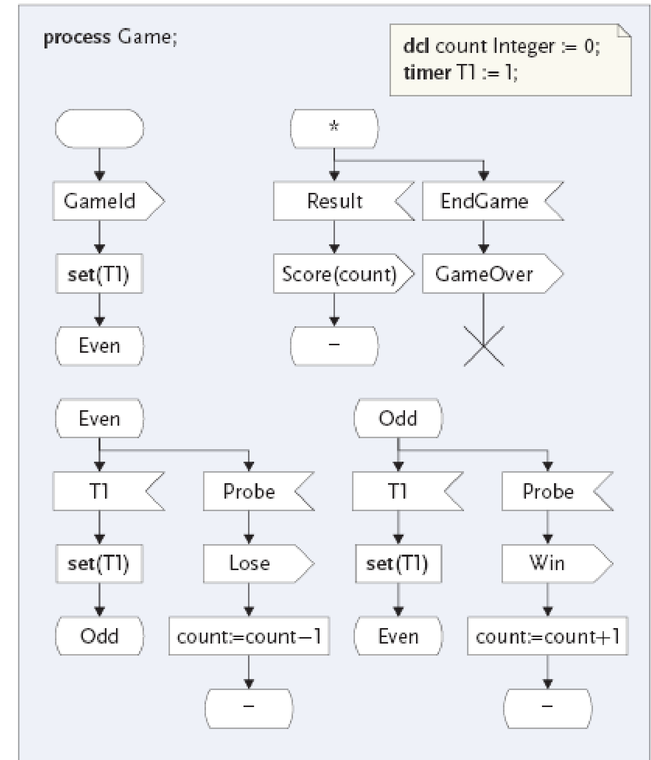
From version U2 to version U3

Back to the foundations

CSP (Hoare 1978)	CCS (Milner 1981)
concurrent processes with no shared memory	
rendezvous (combines synchronization and communication)	
Imperative	functional
explicit assignment of variables	parameter passing only
symmetric sequential composition	action prefix
explicit loops	recursive processes only
no formal semantics	formal semantics (SOS, LTS)

Problems with the CCS style

- Action prefix encourages code duplication
- CCS encodes behaviours as (SDL-like) state machines:
  - many auxiliary processes
  - decision trees ending with process calls
  - multiple calls from a process to another (similar to GOTOs)



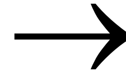
- Instead, we seek concise, readable models
  - reapply "structured programming" ideas to LNT code

Common-sequence merge

- Idea:
  - replace action prefix by LNT's symmetric sequential composition to enable dag-like code sharing
  - factor out duplicated code in **if** / **case** / **alt** branches
  - apply algebraic transformation laws, e.g.:
    alt B1; B [] B2; B **end alt** = **alt** B1 [] B2 **end alt**; B
  - factor out multiple calls to the same process
    (each process P should call process Q at most once)

Example: process N

```
process N [...] (...) is
   var V: BIT in
      RECEIVE_BLOCK_PROPOSAL (?V);
      if NAT (ID) <= H then
         N_PRIME [...] (ID, P_H, HONEST)
      elsif V = 1 then
         N_PRIME [...] (ID, P_H, MALICIOUS)
      else
         N_PRIME [...] (ID, P_H, DISGUISED)
      end if
   end var
end process
```



```
process N [...] (...) is
   var V: BIT, M: MORALITY in
      RECEIVE_BLOCK_PROPOSAL (?V);
      if NAT (ID) <= H then
         M := HONEST
      elsif V = 1 then
          M := MALICIOUS
      else
         M := DISGUISED
      end if;
      N_PRIME [...] (ID, P_H, M)
   end var
end process
```

- Results:
  - fewer lines of LNT code: 320 (U2) → 288 (U3)
  - fewer LNT process calls: 22 (U2) → 9 (U3)
  - fewer reachable states: -16% ... -19%

From version U3 to version U4

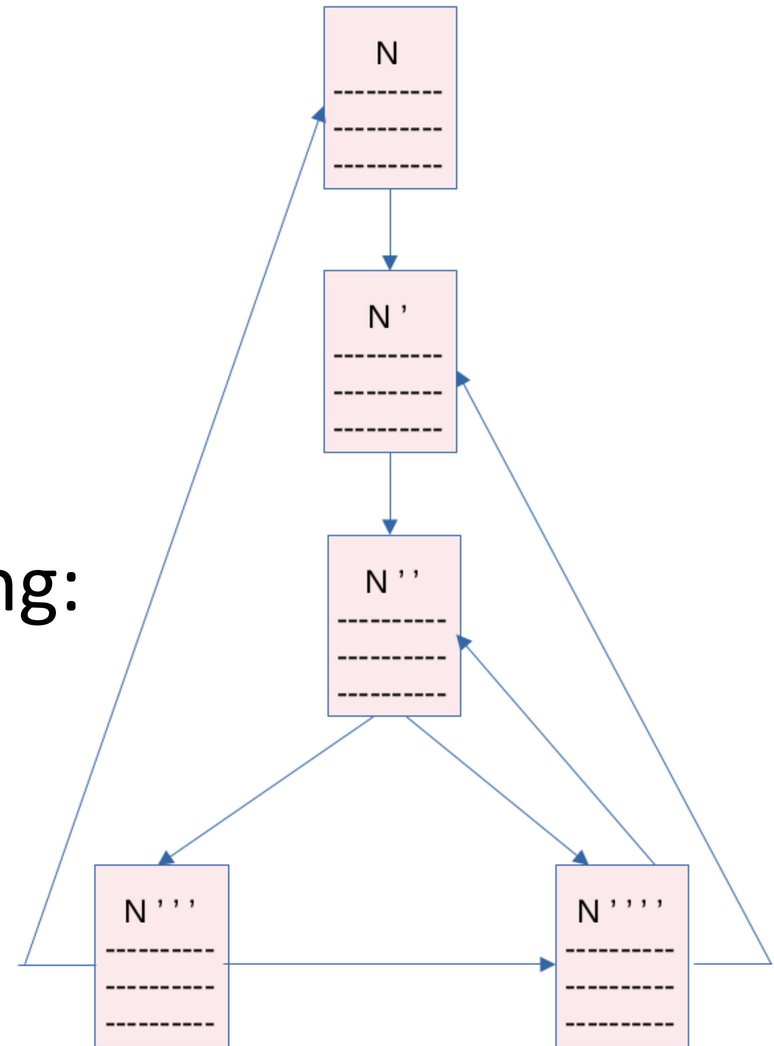
Explicit loops

- Problem:
  - CCS-like tail recursion creates auxiliary processes
  - the execution flow is scattered among many processes (each process call is like a "goto")
- Idea:
  - shift from state machiness to structured programming
  - replace recursive processes by LNT loops (with breaks)
  - more concise and more readable code
  - fewer lines of LNT code: 288 (U3) $\rightarrow$ 250 (U3)

Call graph of node processes N*

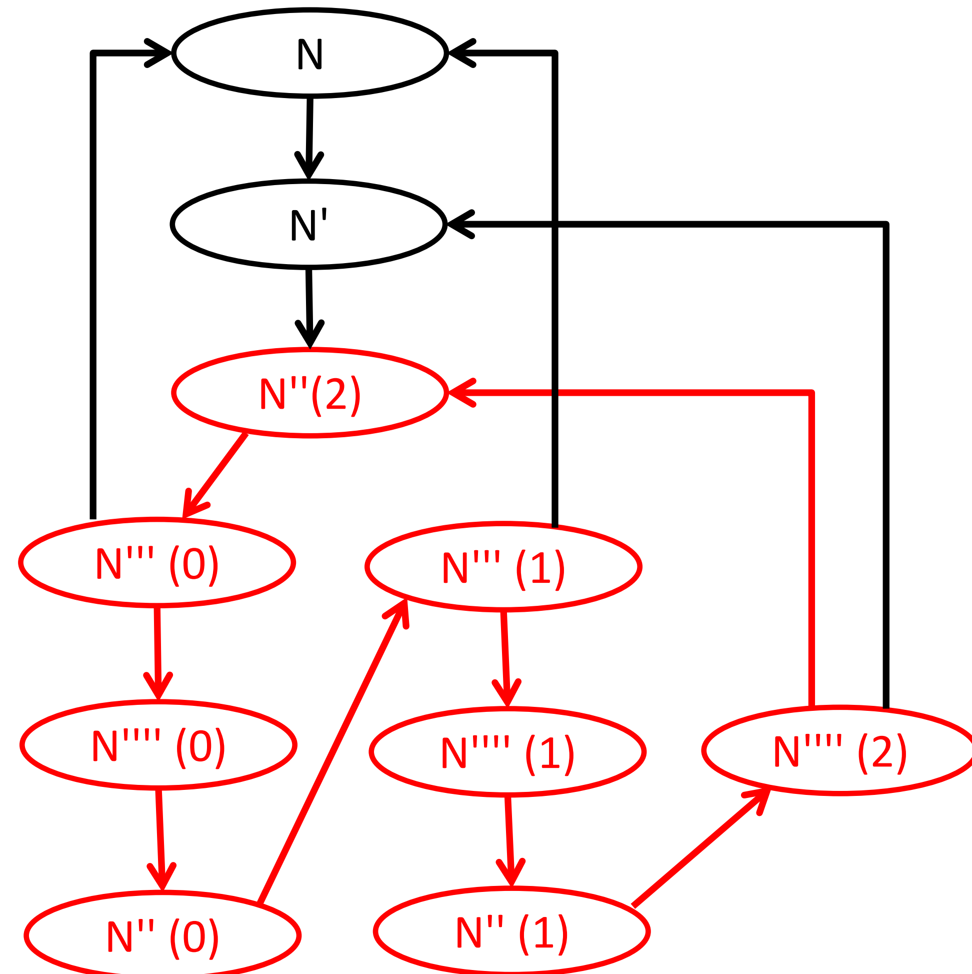
- The NODE process is given as a state machine with 5 "main" states:
  - processes N, N', N'', N''', N''''

- The call graph is disappointing:
  - small, yet intricate
  - no benefit in using LNT loops
  - far from [Chen-Micali-19]



Call graph of processes N* x STEP

- Some N* processes have a STEP parameter (in the range 0, 1, 2)
- Idea:
  - the state is scattered
  - it is not only in N*
  - but in the cartesian product N* x STEP
- Two nested loops appear (rounds, steps)



Explicit loops: solution 1

(S = true)

N

N'

(S = false)

N'' (2)

N''' (0)

N''' (1)

N'''' (0)

N'''' (1)

N'''' (2)

N'' (0)

N'' (1)

```
S := true;
loop
   if S then N;
   N';
   loop L in
      N'' (2);
      if ... then S := true; break L;
      N''' (0); N'''' (0); N'' (0); N''' (1);
      if ... then S := true; break L;
      N'''' (1); N'' (1); N'''' (2);
      if ... then S := false; break L;
   end loop
end loop
```

Explicit loops: solution 2

N

N'

N'' (2)

N''' (0)

N''' (1)

N'

N'''' (0)

N'''' (1)

N'''' (2)

N'' (0)

N'' (1)

process N' is duplicated here

```
loop
   N; N';
   loop L in
      N'' (2);
      if … then break L;
      N''' (0); N'''' (0); N'' (0); N''' (1);
      if … then break L;
      N'''' (1); N'' (1); N'''' (2);
      if … then N'
   end loop
end loop
```

Introduction of 3 auxiliary processes

- Simple, sequential processes
- Reintroduce modularity, in an other way than U3
- Match the textual explanations of [Chen-Micali-19]

```
process BROADCAST [SYNC: SYNCHRONIZE, PROPAGATE: PROPAGATE,
                   SELF_PROPAGATE: SELF_PROPAGATE, TALLY: TALLY,
                   P_IN, P_OUT: PROBABILISTIC]
                  (ID: PID, in var B: BIT, M: MORALITY, out K0, K1: NAT) is
   B := B or BIT (M = MALICIOUS);
   SYNC (BEGIN);
   alt
      P_IN (ID, P_V);
      PROPAGATE (ID, B);
      SELF_PROPAGATE (ID, B)
   []
      P_OUT (ID, 1_MINUS (P_V))
   end alt;
   SYNC (END);
   TALLY (ID, ?K0 where K0 <= N, ?K1 where K1 <= N)
end process

process FLIP_COIN [SET_BIT: SET_BIT, P_ZERO, P_ONE: PROBABILISTIC]
                  (ID: PID, S: STEP, out var B: BIT, P: PROB) is
   alt
      P_ZERO (ID, P);
      B := 0
   []
      P_ONE (ID, 1_MINUS (P));
      B := 1
   end alt;
   SET_BIT (ID, S, B)
end process

process FIX_COIN [SET_BIT: SET_BIT]
                 (ID: PID, S: STEP, out var B: BIT, NEW_B: BIT) is
   B := NEW_B;
   SET_BIT (ID, S, B)
end process
```

The NODE process (version U4)

```
process N [RECEIVE_BLOCK_PROPOSAL, COMMIT_PROPOSED_BLOCK,
        COMMIT_EMPTY_BLOCK: BLOCK, BOYCOTT: BOYCOTT,
        SYNC: SYNCHRONIZE, SET_BIT: SET_BIT,
        PROPAGATE: PROPAGATE, SELF_PROPAGATE: SELF_PROPAGATE,
        TALLY: TALLY, P_IN, P_OUT, P_ZERO, P_ONE: PROBABILISTIC]
       (ID: PID, in var M: MORALITY) is
  var B: BIT, K0, K1: NAT in
    loop
      RECEIVE_BLOCK_PROPOSAL;
      if M <> HONEST then
        alt
          BOYCOTT;
          M := MALICIOUS
        []
          i;
          M := DISGUISED
        end alt
      end if;
      FLIP_COIN [...] (ID, 0, ?B, P_H);
      loop L in
        -- Coin-Fixed-To-0 step
        BROADCAST [...] (ID, B, M, ?K0, ?I
        if K0 >= T then
          COMMIT_PROPOSED_BLOCK;
          break L
        end if;
        FIX_COIN [...] (ID, 1, ?B, BIT (K1 >= T));
        -- Coin-Fixed-To-1 step
        BROADCAST [...] (ID, B, M, ?K0, ?K1);
        if K1 >= T then
          COMMIT_EMPTY_BLOCK;
          break L
        end if;
        FIX_COIN [...] (ID, 2, ?B, not (BIT (K0 >= T)));
        -- Coin-Genuinely-Flipped step
        BROADCAST [...] (ID, B, M, ?K0, ?K1);
        if K0 >= T then
          FIX_COIN [...] (ID, 0, ?B, 0)
        elsif K1 >= T then
          FIX_COIN [...] (ID, 0, ?B, 1)
        else
          FLIP_COIN [...] (ID, 0, ?B, 0.5 of PROB)
        end if
      end loop
    end loop
  end var
end process
```

3. VERIFICATION BY STATE-SPACE EXPLORATION

Verification is a continuum

- LNT supports two main types of analyses
- Static analyses:
  - many checks done at compile time
  - uninitialized variables, bad synchronizations, etc.
- Dynamic analyses:
  - (partial or exhaustive) state-space exploration
  - generation of Labelled Transition Systems (LTSs)
  - many CADP tools for verifying LTSs:
    visual checking, equivalence checking, model checking

State-space exploration

- LNT programs are translated to LTSs
  - all reachable states and transitions are explored
  - information is attached to transitions, not states
  - some transitions (noted "i") are non-controllable
- Limitation: state explosion problem
- But:
  - all LTSs for 4-node Algorand models are small ($\leq$ 43,000 states)
  - LTSs generated for 10-node Algorand in Urbino

LTSs for 4-node ALGORAND

- ALGORAND (4, 0) :
  - 4 honest nodes
  - 20,905 states (version U4)
- ALGORAND (2, 2):
  - 2 honest & 2 malicious nodes
  - 35,059 states (version U4)

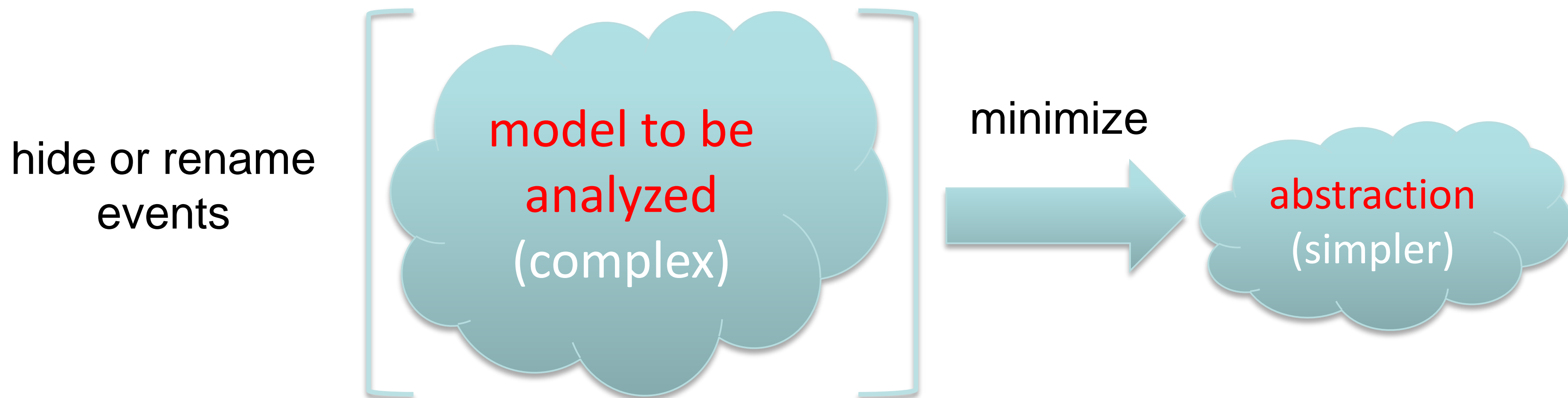
```
131
i
COMMIT_EMPTY_BLOCK
COMMIT_PROPOSED_BLOCK
P_IN !{1...4} !0.75
P_OUT !{1...4} !0.25
P_ONE !{1...4} !{0.6875, 0.5}
P_ZERO !{1...4} !{0.3125, 0.5}
PROPAGATE !{1...4} !{0, 1}
RECEIVE_BLOCK_PROPOSAL !{0, 1}
SELF_PROPAGATE !{1...4} !{0, 1}
SET_BIT !{1...4} !{0...2} !{0, 1}
SYNC !{BEGIN, END}
TALLY* !{1...4} !{0...4} !{0...4}

(*) not all combinations permitted
```

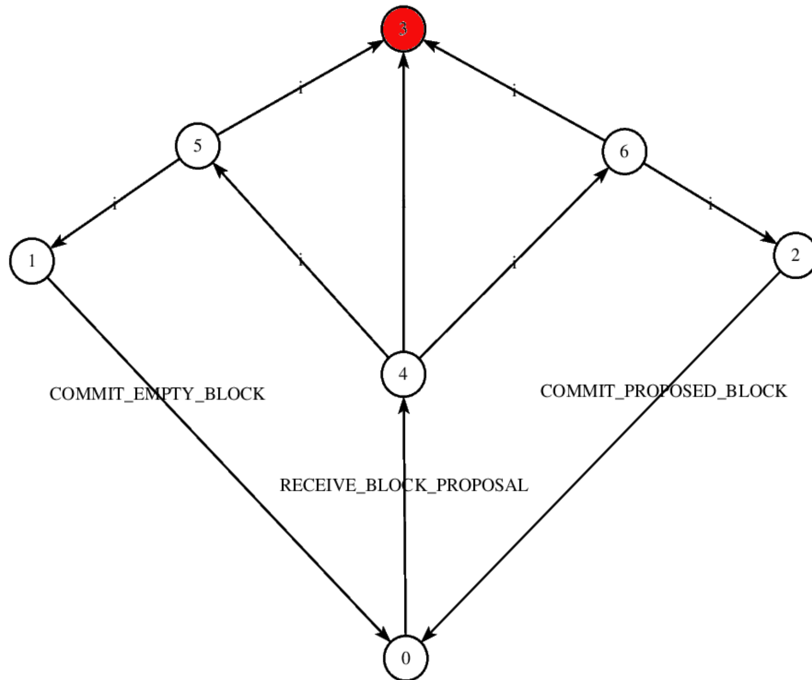
Verification 1: Visual checking

Visual checking

- Generate the LTS
- Event slicing: hide all events but a few ones of interest
- Minimize the LTS for strong or branching bisimulation
- if the minimized LTS if is small enough,
  check its correctness visually



Example: deadlocks



From the initial state, one can reach the 'red' state (deadlock) by different paths

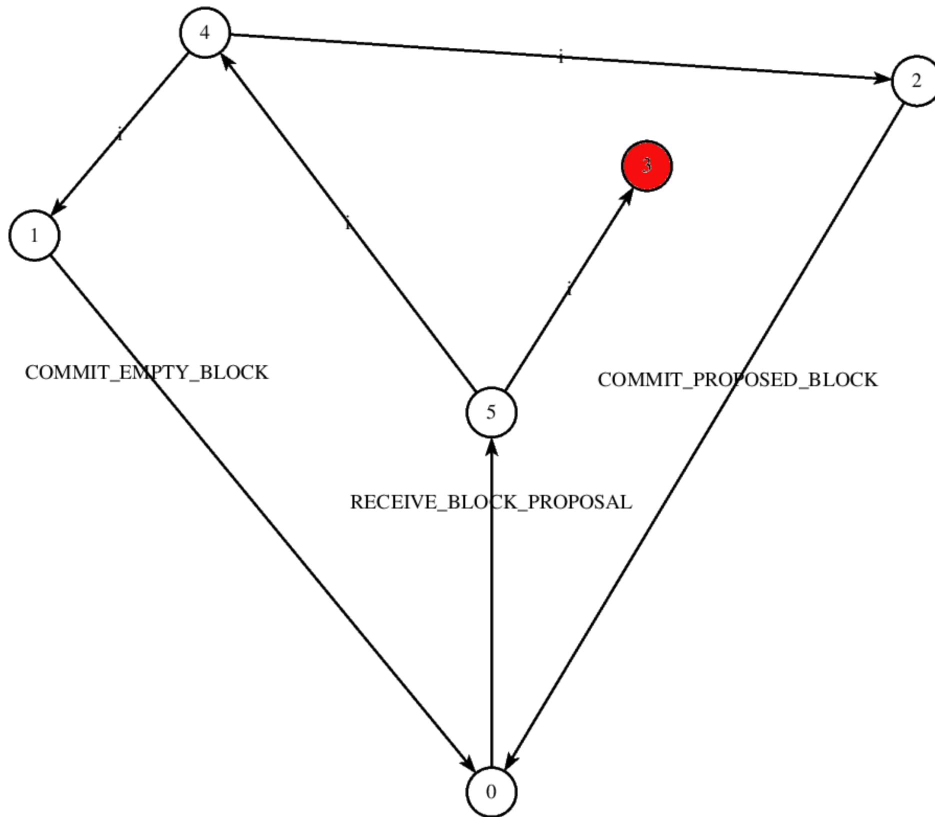
The shortest sequence has 33 transitions

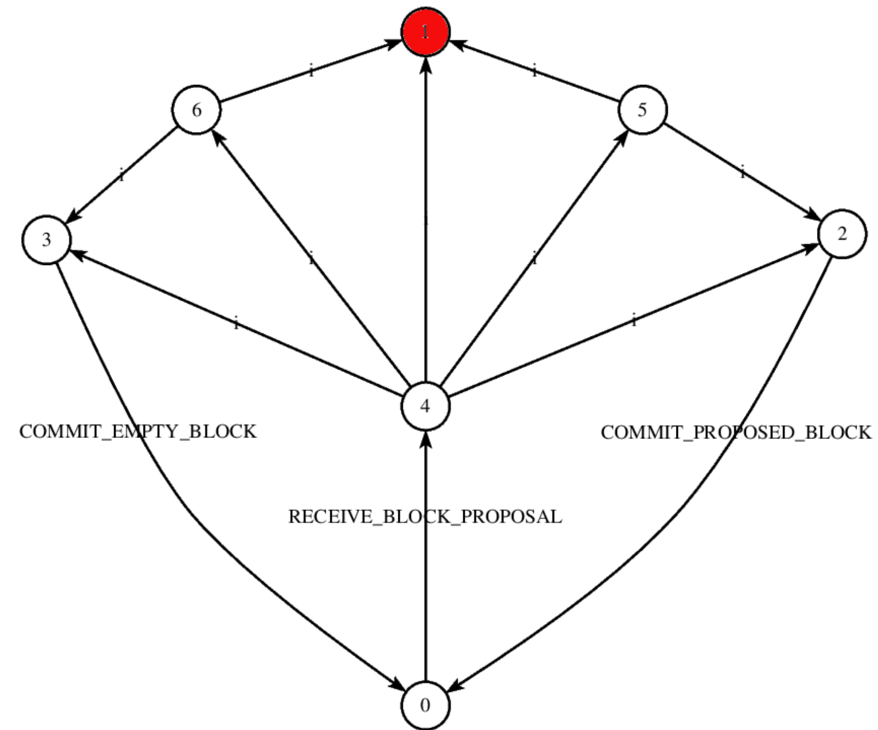
$\Rightarrow$ not easy to detect using classical testing

```
<initial state>
"RECEIVE_BLOCK_PROPOSAL"
"COMPUTE_BIT"
"COMPUTE_BIT"
"P_B !0.2576"
"P_B !0.7424"
"SELF_VERIFY"
"SELF_VERIFY"
"COMPUTE_BIT"
"COMPUTE_BIT"
"P_B !0.2576"
"P_B !0.2576"
"P_OUT !0.25"
"P_IN !0.75"
"SELF_VERIFY"
"P_OUT !0.25"
"SELF_VERIFY"
"PROPAGATE !2 !0"
"P_OUT !0.25"
"SYNC"
"ASK !0"
"REPLY !0"
"ADJUST_BIT"
"ASK !1"
"REPLY !0"
"SELF_VERIFY"
"P_IN !0.75"
"PROPAGATE !2 !0"
"ASK !0"
"ASK !0"
"REPLY !2"
"ASK !0"
"REPLY !2"
"REPLY !2"
<deadlock>
```

Algorand variant with 4 honest nodes
35+ million states, 131+ million transitions
minimized after hiding non-essential events

Other examples of deadlocks

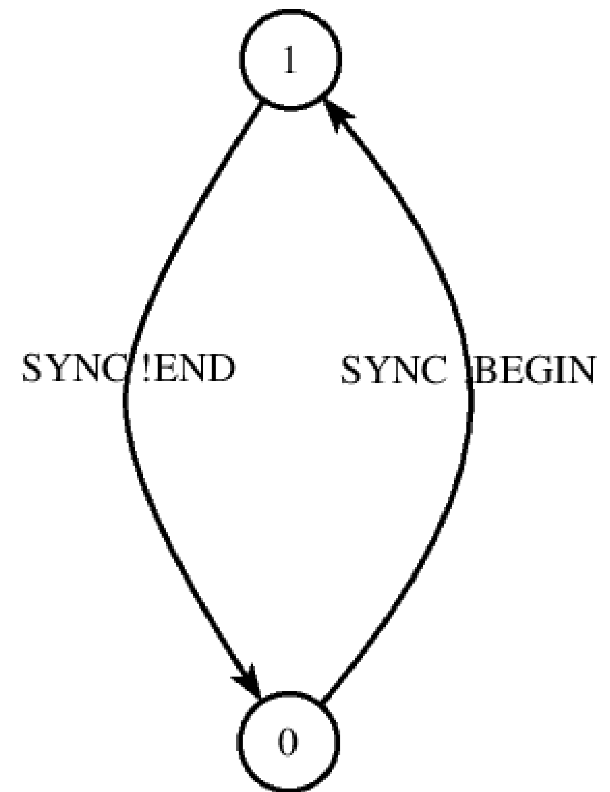




Slicing SYNC events

- We only observe SYNC events
- This can be done easily in SVL

branching reduction in
hide all but SYNC **in**
ALGORAND



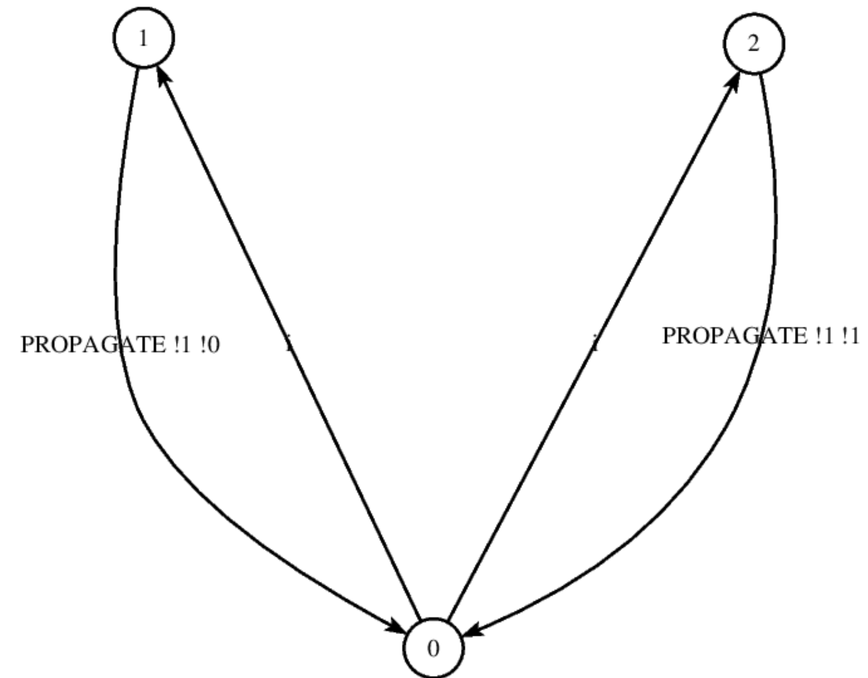
Slicing PROPAGATE$_1$ events

- We observe PROPAGATE events only for node 1
  (all nodes are symmetric)

branching reduction in

total hide all but "SELF_PROPAGATE !1 !.*" **in** ALGORAND

- Same result for the SELF_PROPAGATE$_1$ events



Slicing P_IN_1 and P_OUT_1 events

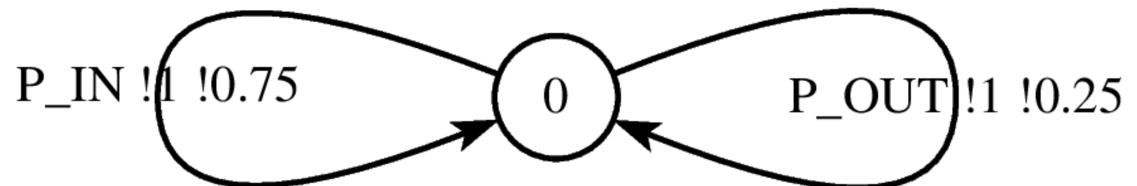
- We observe P_IN and P_OUT events, only for node 1

branching reduction in

total hide all but

"P_IN !1 !.*", "P_OUT !1 !.*" **in**

ALGORAND



Slicing P_ZERO$_1$ and P_ONE$_1$ events

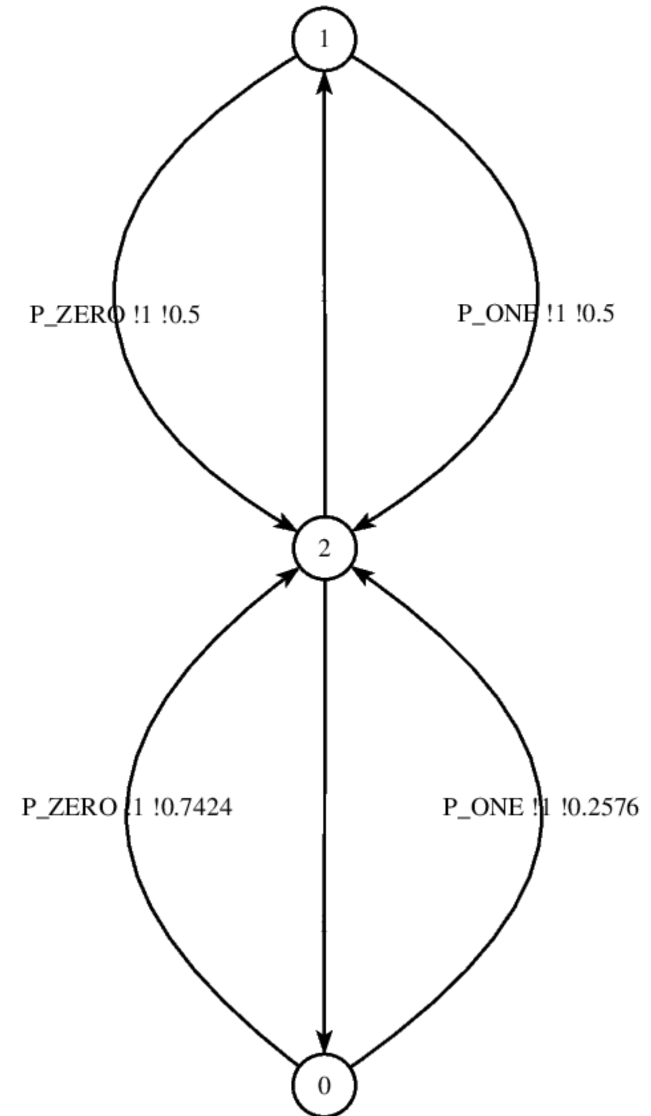
- We observe P_ZERO and P_ONE events only for node 1

branching reduction in

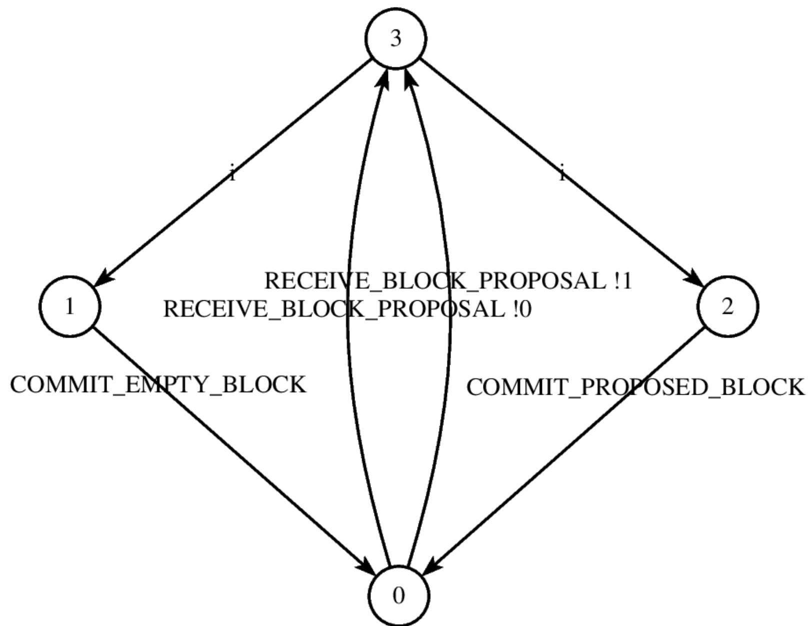
total hide all but

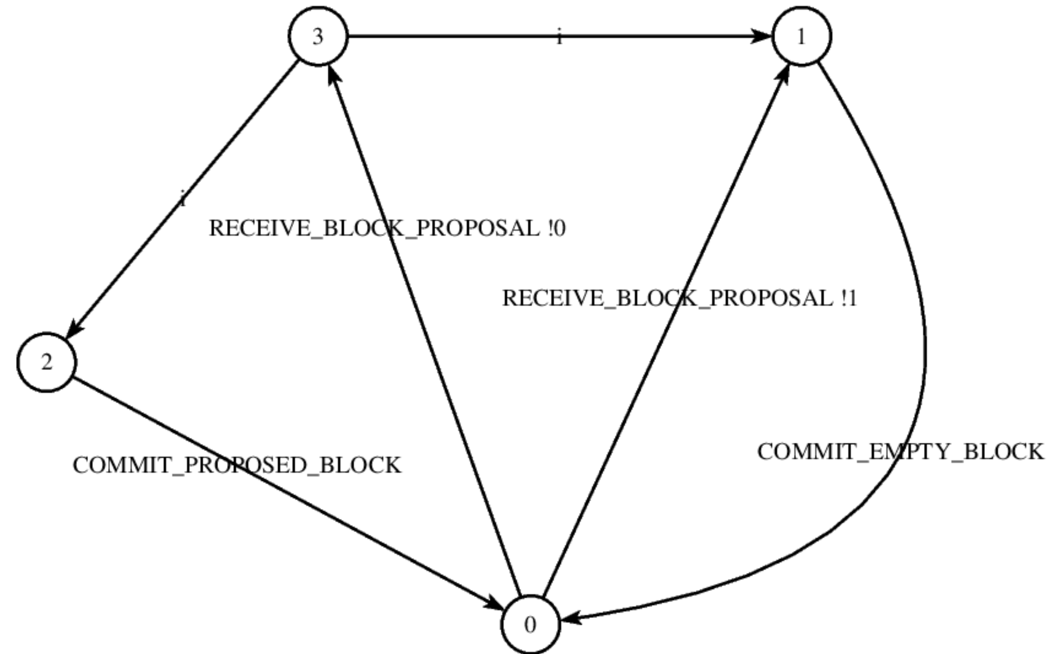
"P_ZERO !1 !.*", "P_ONE !1 !.*" **in**

ALGORAND



Failed vs successful attacks





Algorand with H ≥ T

H : honest nodes
T : threshold value

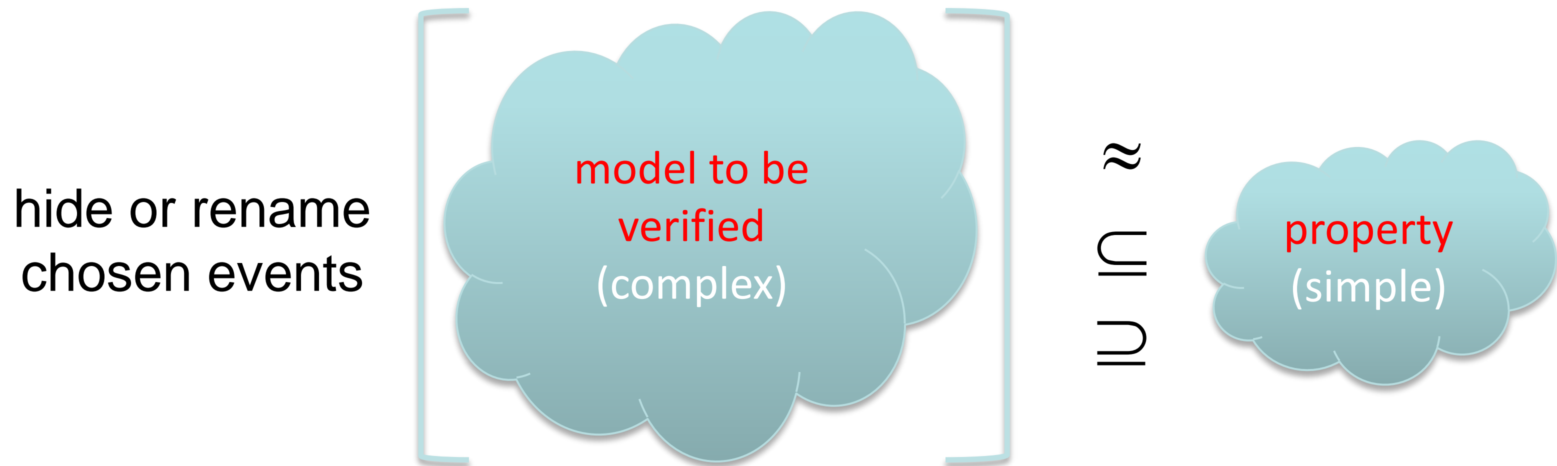
Algorand with H < T

H : honest nodes
T : threshold value

Verification 2:
Equivalence checking

Equivalence checking

- Check whether two LTSs are equivalent (or included)
  - based on bisimulations and preorders (strong, branching, etc.)
  - usually, one compares a complex system to a simple one
  - abstractions: keep only those events of interest



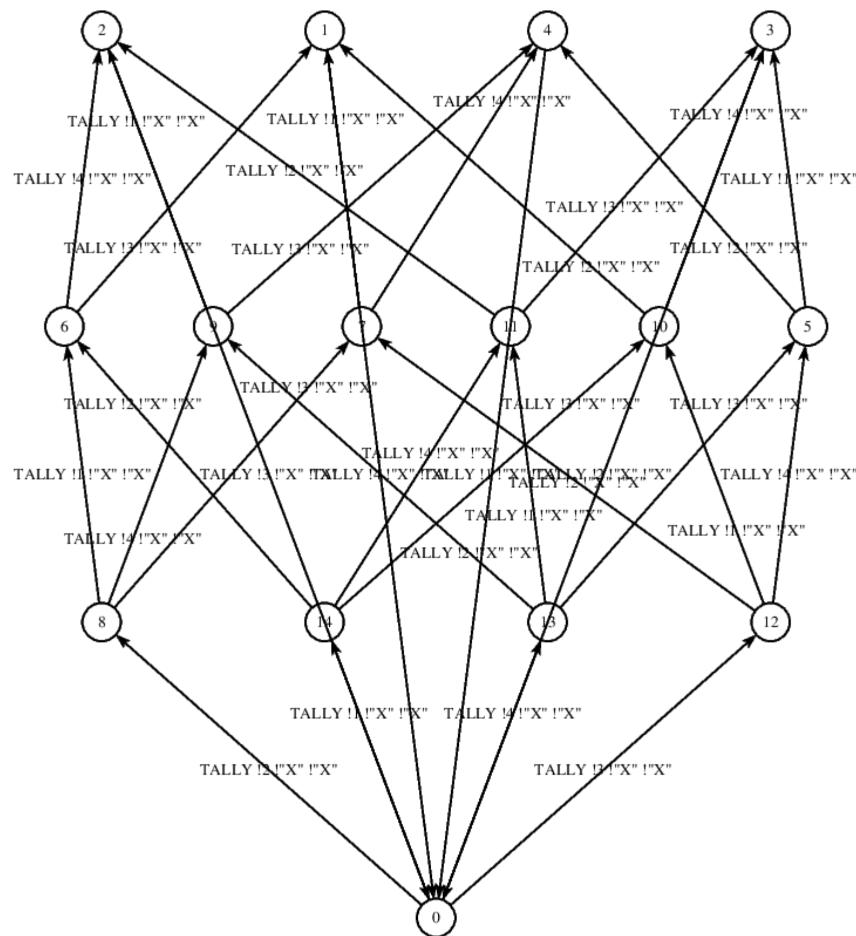
- Also useful to check whether LNT code transformations preserve the semantics

Slicing TALLY events

hide all but TALLY and **rename** TALLY's 2nd/3rd parameters to "X" **in** ALGORAND

$\approx_b$ (branching bisimilar)

```
loop
   par
      TALLY (1 of NAT, "X", "X")
   ||
      TALLY (2 of NAT, "X", "X")
   ||
      TALLY (3 of NAT, "X", "X")
   ||
      TALLY (4 of NAT, "X", "X")
   end par
end loop
```



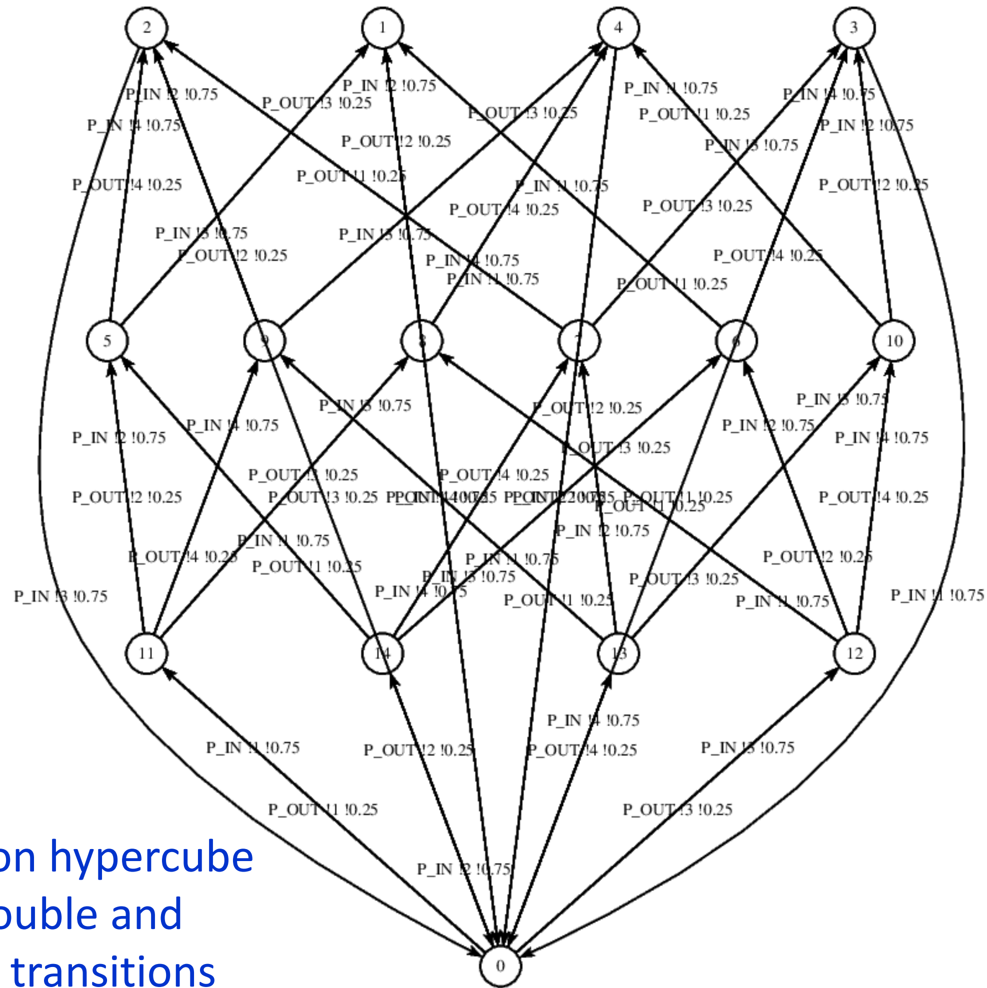
Slicing P_IN and P_OUT events

hide all but P_IN, P_OUT **in** ALGORAND

$\approx_b$

```
loop
   par
      ALT [...] (1)
   ||
      ALT [...] (2)
   ||
      ALT [...] (3)
   ||
      ALT [...] (4)
   end par
end loop

process ALT [P_IN, P_OUT: any] (ID: NAT) is
   alt
      P_IN (ID, 0.75)
   []
      P_OUT (ID, 0.25)
   end alt
end process
```



4-dimension hypercube
with double and
loopback transitions

Slicing P_ZERO and P_ONE events

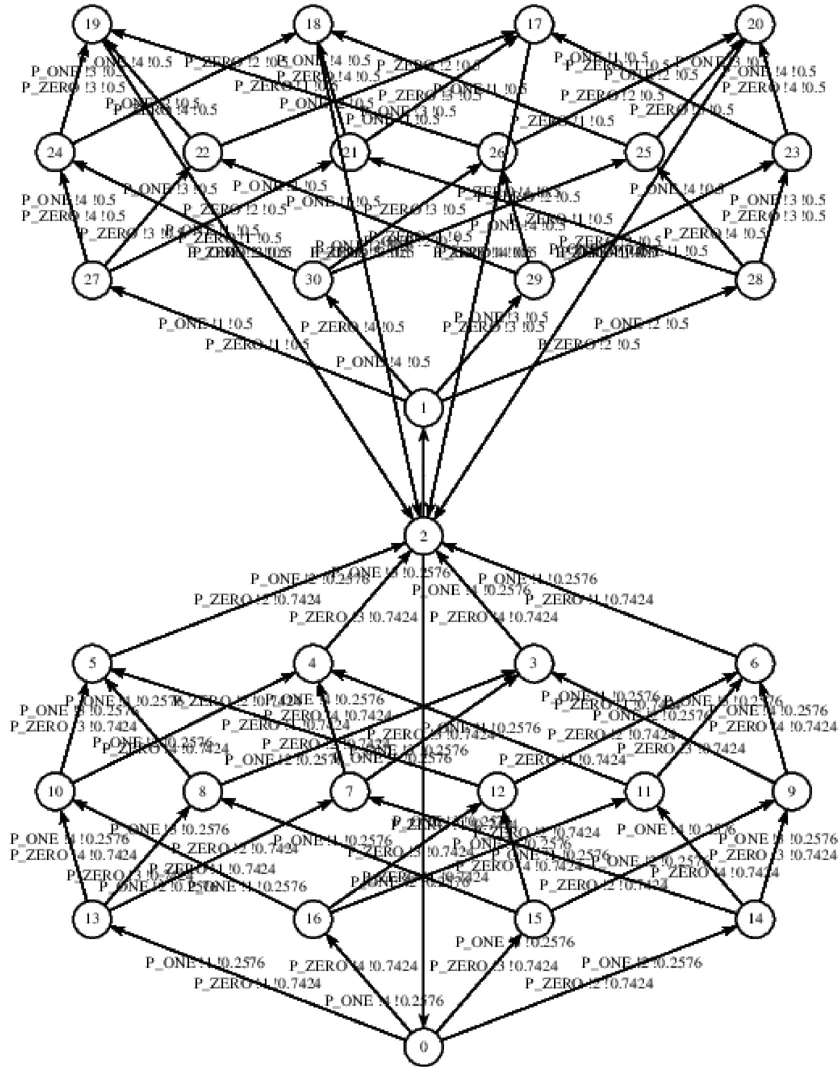
hide all but P_ZERO, P_ONE **in** ALGORAND

$\approx_b$

```
PART [...] (P_H);
loop
   alt
      i; PART [...] (0.5 of PROB)
   []
      i; PART [...] (P_H)
   end alt
end loop

process PART [P_ZERO, P_ONE: any] (P: PROB) is
   par
      ALT [...] (1 of PID, P)
   ||
      ALT [...] (2 of PID, P)
   ||
      ALT [...] (3 of PID, P)
   ||
      ALT [...] (4 of PID, P)
   end par
end process

process ALT [P_ZERO, P_ONE: any] (ID: PID, P: PROB) is
   alt
      P_ZERO (ID, P)
   []
      P_ONE (ID, 1_MINUS (P))
   end alt
end process
```



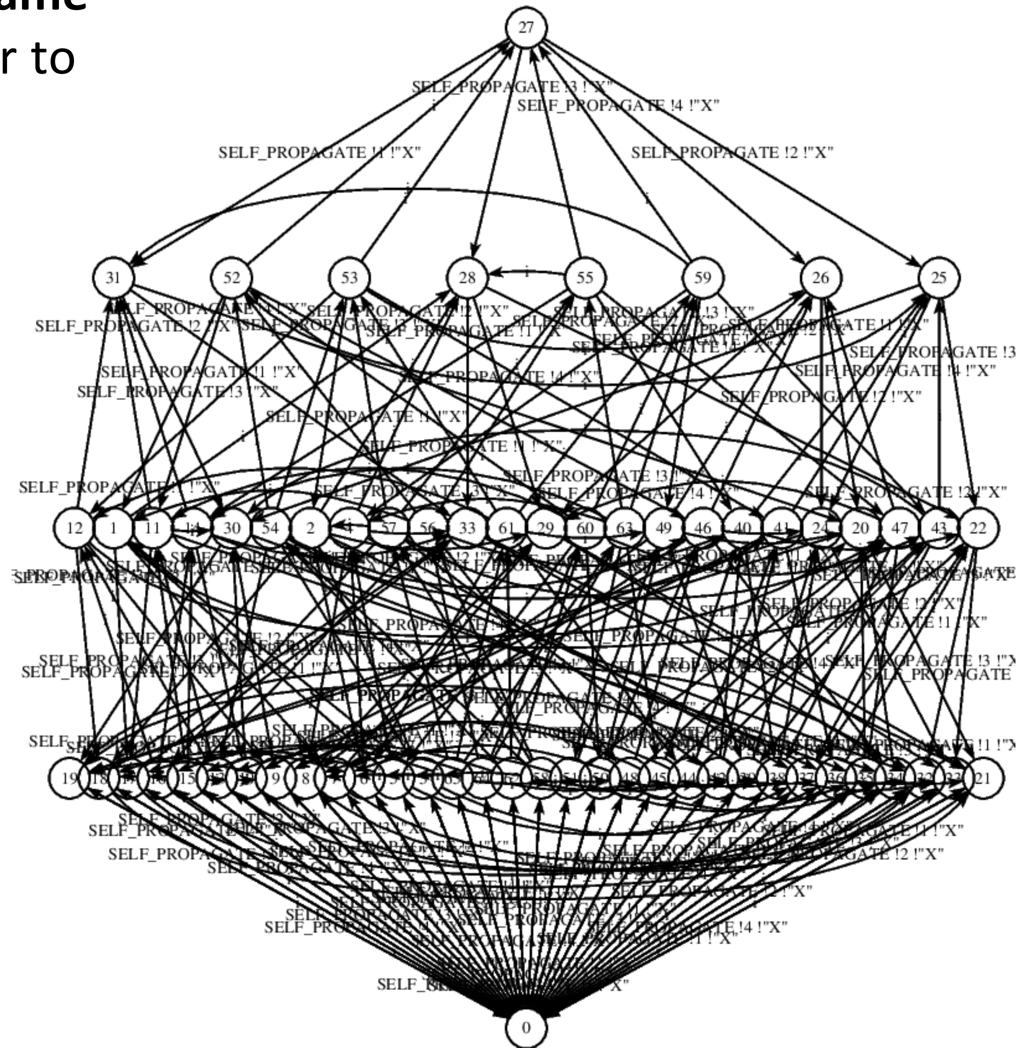
Slicing SELF_PROPAGATE events

hide all but SELF_PROPAGATE and **rename** SELF_PROPAGATE 's 2nd parameter to "X" **in** ALGORAND

$\approx_b$

```
loop
   par
      PART [...] (1)
   ||
      PART [...] (2)
   ||
      PART [...] (3)
   ||
      PART [...] (4)
   end par
end loop

process PART [SELF_PROPAGATE: any] (ID: NAT) is
   i;
   alt
      i; SELF_PROPAGATE (ID, "X")
   []
      i
   end alt
end process
```



Verification 3: Model checking

Model checking

- Visual and equivalence checking may be infeasible
  - when the model is too large — worst-case: O(m log n)
  - when the property is too complex to be expressed as a graph

- Alternative solution: model checking
  - the property is expressed as a set of temporal logic formulas
  - one checks that the model satisfies each of these formulas
  - the model checker answers TRUE or FALSE + counter-examples

- The MCL formula language of CADP combines:
  modal mu-calculus + regular expressions + value-passing extensions

Example 1: TALLY events

MCL formulas can easily express constraints on event parameters

TALLY events have three parameters:

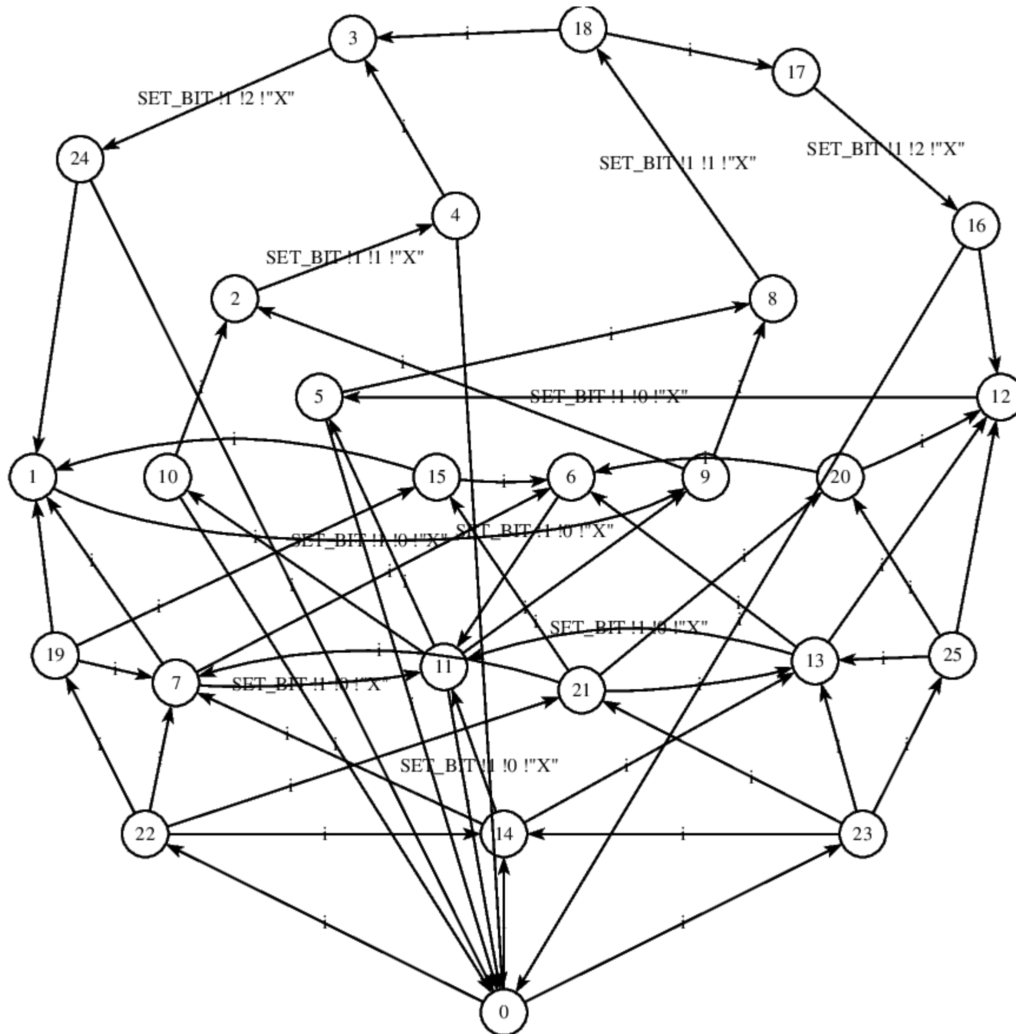
TALLY !ID∈1…4 !K0 !K1

K1 → number of votes for rejection

K0 → number of votes for acceptance

```
property P3a (MODEL)
   "the sum of both TALLY counters is always in the range 0...4"
is
   "$MODEL.bcg" |=
   not (POSSIBLE ({ TALLY ?any ?K0:nat ?K1:nat where K0 + K1 > 4 }));
   expected TRUE
end property
```

Example 2: SET_BIT events



- Slicing SET_BIT events (even for a single node) gives a complex LTS:
  - many "i" events
  - no obvious structure
- Equivalence checking seems impossible:
  - use model checking

Execution traces for SET_BIT events

```
"RECEIVE_BLOCK_PROPOSAL"
"SET_BIT !3 !0 !1"
"SET_BIT !1 !0 !0"
"SET_BIT !2 !0 !1"
"SET_BIT !4 !0 !0"
"SET_BIT !3 !1 !1"
"SET_BIT !2 !1 !1"
"SET_BIT !4 !1 !1"
"SET_BIT !1 !1 !1"
"COMMIT_EMPTY_BLOCK"
"RECEIVE_BLOCK_PROPOSAL"
"SET_BIT !1 !0 !0"
"SET_BIT !3 !0 !1"
"SET_BIT !2 !0 !1"
"SET_BIT !4 !0 !0"
"SET_BIT !4 !1 !0"
"SET_BIT !3 !1 !0"
"SET_BIT !2 !1 !0"
"SET_BIT !1 !1 !0"
"SET_BIT !1 !2 !0"
"SET_BIT !2 !2 !0"
"SET_BIT !4 !2 !0"
"SET_BIT !3 !2 !0"
"SET_BIT !4 !0 !0"
"SET_BIT !1 !0 !0"
"SET_BIT !2 !0 !0"
"SET_BIT !3 !0 !0"
"COMMIT_PROPOSED_BLOCK"
"RECEIVE_BLOCK_PROPOSAL"
...
```

- SET_BIT events have 3 parameters:

  SET_BIT !ID∈1…4 !STEP∈0…2 !BIT∈0…1

- One observes regular patterns:
  - 1st parameter (ID):

    sequences of 4 events (one per Algorand node) in random order
  - 2nd parameter (STEP):

    step numbers increase modulo 3
  - 3rd parameter (B):

    complex rules of consensus making

MCL properties for SET_BIT (1/3)

```
property P9a (MODEL)
   "after RECEIVE_BLOCK_PROPOSAL (any), for each ID in {1...4}, it is inevitable"
   "to reach a SET_BIT (ID, 0, any) event by following a path that contains"
   "neither RECEIVE_BLOCK_PROPOSAL (any) nor SET_BIT (ID, 1 or 2, any)"
is
   "$MODEL.bcg" |=
   forall ID:NAT among {1 ... 4} .
      AFTER_1_WITHOUT_2_INEVITABLE_3 (
         { RECEIVE_BLOCK_PROPOSAL ?any },
         { RECEIVE_BLOCK_PROPOSAL ?any } or { SET_BIT !ID ?any ?any },
         { SET_BIT !ID !0 ?any });
end property
```

MCL properties for SET_BIT (2/3)

```
property P9b (MODEL)
   "for each ID in {1...4}, after SET_BIT (ID, STEP, any), where STEP < 2,"
   "it is inevitable to reach either SET_BIT (ID, STEP + 1, any) or"
   "RECEIVE_BLOCK_PROPOSAL (any) by following a path that contains neither"
   "RECEIVE_BLOCK_PROPOSAL (any) nor SET_BIT (ID, any, any)"
is
   "$MODEL.bcg" |=
   forall ID:NAT among {1 ... 4} .
      AFTER_1_WITHOUT_2_INEVITABLE_3 (
         { SET_BIT !ID ?STEP:NAT ?any where STEP < 2 },
         { RECEIVE_BLOCK_PROPOSAL ?any } or { SET_BIT !ID ?any ?any },
         { RECEIVE_BLOCK_PROPOSAL ?any } or { SET_BIT !ID !STEP + 1 ?any });
end property
```

MCL properties for SET_BIT (3/3)

```
property P9c (MODEL)
   "for each ID in {1...4}, after SET_BIT (ID, 2, any), it is inevitable"
   "to reach either SET_BIT (ID, 0, any) by following a path that contains"
   "neither RECEIVE_BLOCK_PROPOSAL (any) nor SET_BIT (ID, any, any)"
is
   "$MODEL.bcg" |=
   forall ID:NAT among {1 ... 4} .
      AFTER_1_WITHOUT_2_INEVITABLE_3 (
         { SET_BIT !ID !2 ?any },
         { RECEIVE_BLOCK_PROPOSAL ?any } or { SET_BIT !ID ?any ?any },
         { SET_BIT !ID !0 ?any });
end property
```

4. CONCLUSION

Results

- LNT was designed to describe large systems
  When used properly, it allows concise models

- Illustrated with Algorand BBA* consensus protocol
  - generic transformations to reduce LNT code size
  - five formal models U0 … U4, all strongly bisimilar
  - verification by visual/equivalence/model checking

- A lot of work (hidden under the hood):
  - 420+ LNT models of BBA* developed
  - enhancements to the LNT language and its compilers

Future work

- Verify more properties of BBA* consensus
  - ongoing work in Grenoble (analysis of SET_BIT events)
- Perform probabilistic verification of BBA*
  - ongoing work in Urbino
- Relax the synchrony assumption (SYNC events)
  - extend to a "substantially asynchronous" setting
- Study ABFT (Algorand Byzantine Fault Tolerance)
  - the evolution of BBA* currently used in Algorand