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

Hubert Garavel
Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
hubert.garavel@inria.fr

LNT is a modern language for the formal description of concurrent systems. It generalizes traditional process calculi and overcomes their known limitations by incorporating features such as an imperative programming style with direct assignments to variables, symmetric sequential composition, and explicit loop operators. The present article examines how these features can be taken advantage of to obtain LNT models as concise and readable as possible. The study is illustrated with a running example, the consensus protocol of the Algorand blockchain, a formal model of which was recently developed at the University of Urbino. It is shown that, using well-chosen transformations, the number of lines of LNT code can be divided by three, while improving readability. Also, various properties of the formal model are expressed and verified using visual checking, equivalence checking, and model checking.

1 Introduction

Concurrent systems are difficult to design and evolve without the guidance of formal methods and the assistance of verification tools. LNT[1] is a modern language for the formal description of such systems. Inspired by LOTOS [21] [3], Occam [26] [18], and E-LOTOS [22], LNT pursues three main objectives:

1. Achieving a synthesis between, on the one hand, process calculi, which have been proposed to model concurrent systems, and, on the other hand, mainstream (imperative or functional) programming languages, which are routinely used to develop sequential programs;
2. Being supported by robust software tools, such as TRAIAN [28], a compiler front-end (and C-code generator) that performs involved static analyses to detect design mistakes in LNT models as early as possible, and LNT2LOTOS [15] [6], a translator from LNT to LOTOS that enables one to explore the state spaces (Labelled Transition Systems, LTSs for short) of LNT programs and verify them using the equivalence checkers and model checkers of the CADP toolbox [13]. To our knowledge, no other modelling language for concurrent systems benefits from such a wide range of static and dynamic analyses as available for LNT.
3. Scaling properly to complex models of real systems, which is achieved, on the language side, by equipping LNT with programming-in-the-large features (such as modules, time-proven constructs for structured programming, Ada-like bracketed syntax, etc.) and, on the software side, by making sure that the LNT compilers can handle involved programs (for instance, the TRAIAN and LNT2LOTOS tools are themselves written in LNT, totalling 94,000 non-blank lines of LNT code).

Since LNT is being used in an increasing number of case-studies[2] and research tools[3], the question often arises of how to write LNT code concisely. Indeed, the versatility of LNT makes it possible to write the

[1] https://cadp.inria.fr/tutorial/index.html#lnt
[2] https://cadp.inria.fr/case-studies
[3] https://cadp.inria.fr/software

Maurice H. ter Beek and Gregor Gössler (Eds.):
Proceedings of the 7th Workshop on Models
for Formal Analysis of Real Systems (MARS'26)
EPTCS 443, 2026, pp. 43–83, doi:10.4204/EPTCS.443.5

© H. Garavel

same fragment of code in many diverse ways. The present paper addresses this question, reviewing the different options, and establishing comparisons with traditional process calculi.

Throughout the present article, we use Algorand as a running example. Algorand is a high-performance Layer-1 blockchain that brings convincing proposals in terms of decentralization, scalability, security, and low-energy consumption. It is promoted by a foundation[4] and a company[5], and supports two cryptocurrencies (ALGO and EURD). At the heart of Algorand is a secure distributed ledger algorithm proposed by Chen & Micali [7], a crucial part of which is the BBA* (generalized Binary Byzantine Agreement) consensus protocol. A detailed, user-friendly presentation of Algorand can be found in [9], which also proposes a formal model of BBA* expressed, first, using process-algebraic notations and, then, using LNT. We build upon that latter model, the goal being to make it simpler and more concise through a series of incremental transformations.

The present article is organized as follows. Section 2 introduces the successive LNT models of Algorand consensus protocol. Sections 3, 4, 5, and 6 present four "generic" techniques for reducing the size of LNT models. Section 7 indicates how these models can be formally verified using state-space exploration techniques. Section 8 gives concluding remarks and suggests directions for future work.

2 Formal Modelling of Algorand Consensus Protocol

The starting point for formal models in LNT is the original description of Algorand [7], which (unlike many other blockchains) was not merely a white paper, but a published article in an established scientific journal. Yet, even if this article was precisely written and peer reviewed, the description given of Algorand is informal, contains ambiguities (see Annex A.1), and is not machine-checkable.

The next step was the pioneering work done at the University of Urbino to convert the informal description given by Chen & Micali into a formal, process-algebraic model [9] with various simplifying abstractions (see Annex A.2). An LNT version of this formal model was produced, which, after a few iterations[6], passes all the static checks of the LNT compiler and satisfies basic properties, such as the absence of deadlocks.

2.1 Specific Transformations

Starting from the formal model in LNT given in [9], we derived a new model (named version U0) by devising a number of syntactic and semantic transformations, among which:

- improvement of spacing, indentation, and comments;
- renaming of identifiers to get more concise or more intuitive names;
- introduction of dedicated types for node identifiers and probability values;
- introduction of various functions corresponding to the numeric constants of [7];
- tagging of events with observable information (node identifiers, protocol steps, bit values, etc.);
- removal of some events meant for observation, but later found not so suitable for verification;
- displacement at better locations of some events meant for observation;

[4] `https://algorand.co`

[5] `https://algorandtechnologies.com`

[6] This explains the successive versions v1–v5 of the arXiv report [9]. The present article relies on the final version v5.

- merging of two different, yet similar events into a single event;
- splitting of a given event into two events to avoid nondeterminism;
- adoption of a different attacker model that tries to reject blocks carrying the bit one.

These transformations, detailed in Annex A.3, are called "specific" because they largely depend on the formal model of Algorand and are not directly reusable for other applications. Many of these transformations modify the transitions or transition labels of the LTSs generated from the LNT code, and thus do not preserve bisimilarity properties.

2.2 Generic Transformations

Reducing the size (measured in lines of code) of formal models is a suitable goal. It is expected to reduce the effort spent in reading and understanding these models, correcting their mistakes, maintaining them over time, and implementing them to produce real systems.

In this section and the next ones, we focus on transformations that we call "generic" as they do not modify the semantics of LNT models. Precisely, such transformations should preserve strong bisimilarity between the models before and after transformation.

A simple generic transformation to reduce the size of formal models consists in removing useless definitions, i.e., all objects defined but not used (this is likely to occur when a model undergoes many successive evolutions). The TRAIAN compiler for LNT helps to detect such cases of over-specification by warning about all types, variables, functions, processes, etc. that are not actually useful.

To reduce the size of formal models, other generic transformations exist. Four of them are presented in the next Sections 3 to 6 and applied to version U0 of the Algorand consensus protocol, leading to four successive versions noted U1, U2, U3, and U4. Thus, the five LTSs generated for versions U0 to U4 are expected to be strongly bisimilar. Using the BCG_CMP tool[7] of CADP, we verified this property on the two Algorand configurations $A_{4,0}$ and $A_{2,2}$ studied in [9], where $A_{H,M}$ denotes a network with H honest nodes and M malicious nodes.

The table below gives comparative information about the five versions U0 to U4, both in lines of code for the various LNT models and in numbers of states for the corresponding LTSs (generated using CADP). In this table, L is the total number of LNT lines; ℓ is the total number of LNT lines, excluding blank lines and comments; $S_{1,0}$ is the number of states of the LTS generated for one honest node; $S_{0,1}$ is the number of states of the LTS generated for one malicious node; $S_{4,0}$ is the number of states of the LTS generated for $A_{4,0}$; and $S_{2,2}$ is the number of states of the LTS generated for $A_{2,2}$. The last line of the table gives the numbers of states for these four LTSs minimized wrt strong bisimulation, these numbers being identical for all five versions.

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

[7] https://cadp.inria.fr/man/bcg_cmp.html

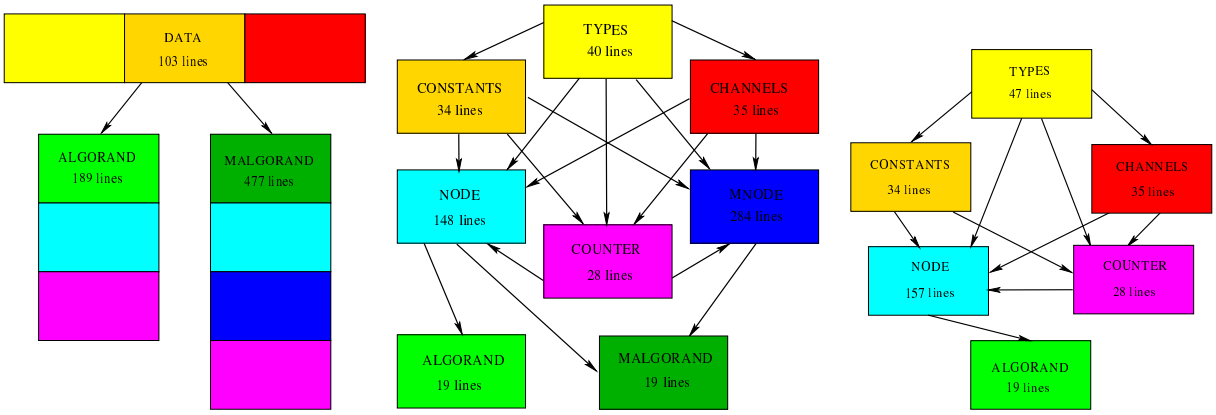


Figure 1: Modular decompositions of versions U0 (left), U1 (middle), and U2 (right)

3 Model Compaction Through Modules

Quite often, formal models contain identical code fragments that the specifier replicated using cut-and-paste facilities. Also, various instances of a formal model may contain identical code fragments located in different computer files. Such a duplication of code is a true nuisance: it is easy to create, but has enormous costs on the long run in terms of bug fixes and maintenance.

Modularity is a proper way to fight code duplication. Pioneering programming languages (e.g., Ada and Standard ML) offer sophisticated modules with interfaces and generics. Similar ideas [5] [4] [29] have been put forward for the design of E-LOTOS, but never actually implemented.

LNT currently proposes a lighter approach to modules, which addresses most practical needs. Each LNT module is a file containing definitions of types, functions, processes, and/or channels[8]. Modules may import other modules: the import relation may be tree-like or dag-like, but may not contain cycles. Each module is self-contained, meaning that it may only refer to objects defined in itself, in the modules it imports, or in the predefined LNT library. Name clashes are forbidden, but this constraint is alleviated by the existence of separate name spaces and the possibility to define overloaded functions. Finally, LNT provides for generic modules parameterized by "virtual" types, functions, processes, and/or channels.

Modules were effective in reducing the size of Algorand models. Version U0 was already decomposed in three modules: `ALGORAND.lnt` and `MALGORAND.lnt`, which describe the configurations $A_{4,0}$ and $A_{2,2}$, respectively, and `DATA.lnt`, which defines types and functions used by these two modules. Yet, as shown in Fig. 1 (left), the two former modules contain two identical fragments of LNT code, namely the definition of a honest Algorand node (light-blue box) and the definition of a counter process (purple box). This prompted for the introduction of two modules that avoid such duplication of code. Additionally, module `DATA.lnt` was split in three smaller modules (types, constants, and channels) and a new module `MNODE.lnt` was created to contain the definition of a malicious Algorand node. This led to version U1, shown in Fig. 1 (middle), where `ALGORAND.lnt` and `MALGORAND.lnt` remain as two tiny modules (19 lines each) that express the top-level architecture. When evolving from version U0 to version U1, the number of lines of LNT code was reduced by more than 20%, while the sizes of LTSs remained unchanged (see Table in Sect. 2.2).

[8] Channels specify the types of communication events.

4 Model Compaction Through Control/Data Tradeoffs

Sequential programs often contain multiple fragments of code that, although not, strictly speaking, identical, are largely similar, differing only in a few details. To reduce the size of such programs and make them easier to maintain, the traditional approach is "procedural abstraction", which encapsulates replicated fragments of code within reusable procedures or functions equipped with appropriate parameters that take care of the few differences between code fragments.

The same approach applies to formal models of concurrent systems, with the additional fact that executing one particular code fragment is an information that plays a role in the global state of the system (as analyzed by explicit-state and symbolic verification). Indeed, the global state usually contains both control information (e.g., location of the program counter, current states in automata, marked places of a Petri net, etc.) and data information (e.g., current value of state variables). In this respect, procedural abstraction not only factors out replicated code fragments, but also converts control information to data information in the global state. For instance, if B_0 and B_1 are two similar LNT code fragments, the nondeterministic choice "**alt** B_0 □ B_1 **end alt**" may equivalently be written "i := **any** *BIT* ; B_i" using nondeterministic selection of a bit value, thus converting control information (program counter located in either B_0 or B_1) into data information (the value of bit i).

Replicated code fragments can be found in version U1 of the Algorand model: the `NODE.lnt` module contains six processes (*NODE*, N, N', N'', N''', and N'''') describing a honest node, whereas the `MNODE.lnt` module contains six processes (*MNODE*, *MN*, MN', MN'', MN''', and MN'''') describing the offensive behaviour of a malicious node and four processes (HN', HN'', HN''', and HN'''') describing the neutral behaviour of a malicious node. Procedural abstraction removes the need for `MNODE.lnt` by merging these processes as follows: {*NODE*, *MNODE*}, {N, *MN*}, {N', MN', HN'}, {N'', MN'', HN''}, {N''', MN''', HN'''}, and {N'''', MN'''', HN''''} after giving them an additional parameter M that may take three values (*honest*, *malicious*, or *disguised*) and, when equal to *malicious*, forces nodes to cast rejection votes. Rather than merging and modifying these processes all at once, with a high risk of errors, the changes were done incrementally, by applying a dozen simple transformations in sequence and checking that strong bisimilarity was preserved at each step.

The model can be further simplified by noticing that, even if each node is assigned a distinct number *ID*, this number has no real impact on the behaviour of honest and malicious nodes (this is related to the ideas of data independence and symmetry reduction of [19]). Without loss of generality, one can assign the lowest numbers to honest nodes, so that comparing the *ID* parameter of each node against the number H of honest nodes determines if the node is honest or not. Thus, module `MALGORAND.lnt` is no longer useful and can be eliminated, leading to the version U2 shown in Fig. 1 (right). With respect to version U1, the number of lines of LNT code in version U2 was divided by two (see Table in Sect. 2.2).

5 Model Compaction Through Common-Sequence Merges

Another cause of verbosity in formal models of concurrent systems lies in the particular history of process calculi. Initially, the pioneering CSP language defined by Hoare [17] in 1978 had the usual (i.e., symmetric) sequential composition operator (noted ";") that already existed in most imperative programming languages. A major shift occurred two years later when Milner proposed its CCS calculus [27], in which sequential composition had to be expressed with an asymmetric "action prefix" operator. Theoretically, CCS offered some advantages: a tiny syntax, a concise semantics, and a sufficient expressiveness. But it had many practical drawbacks: (i) it did not scale well to large systems — actually, few significant

case-studies have been done using CCS; (ii) its sequential and parallel operators fragmented the scientific community into separate schools of thought, e.g., those who adopted action prefix, those who rejected it, and those who tried to have both action prefix and symmetric sequential composition in the same language; (iii) it created a gap between process calculi and conventional programming languages, durably isolating concurrency theory from practical applications and letting lower-level formalisms, such as state machines, take over process calculi.

With respect to model compactness, the action prefix operator of CCS naturally leads to duplicated code fragments, as sequential processes must have a tree-like structure (based on action prefix and nondeterministic choice) with no possibility of dag-like factorization. For instance, "$(b_1 + b_2).B$" is forbidden in CCS and must be written "$(b_1.B) + (b_2.B)$" instead.

The LNT language has a value-passing sequential composition operator that can express the action prefix of CCS as a particular case [10, Sect. 5.3]. When models written in the CCS style are directly translated to LNT, duplicated code fragments are likely to be present, e.g., "**alt** b_1; B □ b_2; B **end alt**" (same for "**if** ... **end if**" and "**case** ... **end case**" statements). Fortunately, the sequential composition operator of LNT allows duplicated code fragments to be factored out, which was not permitted in CCS.

Version U2 of the Algorand model uses the CCS style, since this LNT program was derived from the process-algebraic specification presented in [9]. Hence, there are duplicated sequences of code and recursive process calls in version U2. To reduce their number, each of the five processes N, N', ..., N'''' of `NODE.lnt` was modified separately, using the following transformations (the corresponding code fragments of versions U2 and U3 can be found in Annexes B.7 and B.8, respectively):

- For process N: the transformation is trivial. It introduces an auxiliary variable M and replaces the three calls to process N' by a single call.
- For process N': the transformation is also trivial. It introduces an auxiliary variable B and replaces the two calls to process N'' by a single call
- For process N'': the transformation requires two successive steps. First, each of the three "**alt**" statements is simplified by applying the following law:[9]

  $$\textbf{alt}\ B_1; B\ \square\ B_2; B\ \textbf{end alt}\ =\ \textbf{alt}\ B_1\ \square\ B_2\ \textbf{end alt}\,; B$$

  where "=" denotes strong bisimilarity and where the common suffix B corresponds to the code fragments starting at "SYNC (END)". Then, the three branches of the modified "**case**" statement are simplified by applying the following law:

  $$\textbf{case}\ 0 \to B; B_0 \mid 1 \to B; B_1 \mid 2 \to B; B_2\ \textbf{end case}\ =\ B; \textbf{case}\ 0 \to B_0 \mid 1 \to B_1 \mid 2 \to B_2\ \textbf{end case}$$

  where the common prefix B corresponds to the code fragments starting at "**alt**" and ending with "**end alt**; SYNC (END)". After these transformations, process N'' is still called twice in version U2; it could be factored out, but the resulting code would be longer and less readable.
- For process N''': the transformation exploits the fact that the two executable branches of the "**case**" start with a common prefix (the TALLY event) followed by two largely similar code fragments.
- For process N'''': the transformation exploits the fact that the three branches of the "**case**" bear similarities: the two first branches are dual of each other, and the third branch borrows code from the two other branches. The transformation introduces two auxiliary variables *DONE* and B and replaces the six calls to process N'' by a single call. The resulting code is concise, yet harder to

[9] Notice that this law only holds for common suffixes and would not hold for common prefixes.

understand (its correctness can be checked by executing symbolically both versions of process N'''' three times, one per value of S). The transformation can be seen as an exercise to get a different view at the code of process N'''', possibly suggesting further refactoring of the whole node process.

Again, these transformations were done one at a time, carefully checking the preservation of strong bisimilarity after each change. Between versions U2 and U3, the number of process calls in `NODE.lnt` was reduced from 22 to 9. The number of lines of LNT code was reduced by 10%, with a noticeable effect on state spaces: 16% less states for $S_{4,0}$ and 19% less states for $S_{2,2}$ (see Table in Sect. 2.2).

6 Model Compaction Through Explicit Loops

Our last transformation addresses two causes of superfluous verbosity in models of concurrent systems:

- Communication protocols are often described using state machines, which are defined using control states, transitions between these states, and state variables, whose values are checked and modified by code fragments written in imperative programming languages (as, e.g., in Estelle [20] or SDL [23]). Unfortunately, we know from Dijkstra that gotos can be harmful [8] and, indeed, state machines may easily lead to "spaghetti code", in which the control flow is difficult to follow, especially because loops are not identified explicitly.
- The programming style proposed by Milner for CCS (and later adopted by process calculi such as ACP [2] and LOTOS [21]) radically differs from that of CSP in two more points: (i) CCS has no loop operator, so that any iteration must be expressed using process recursion; (ii) CCS is strictly functional, without explicit assignments to variables. Consequently, in the CCS style, protocols can only be specified using state machines, in a verbose and non-intuitive manner. Precisely, each state machine with n states is encoded by n processes, and each transition from state s_1 to state s_2 is encoded by a tail-recursive call of the process corresponding to s_2. Moreover, each state variable of this machine is declared n times (as a parameter of each of these processes), and n^2 parameter passings are needed to encode the assignments to state variables, even in the cases where these variables are not modified[10].

To improve the quality and compactness of formal models, it is thus desirable to migrate away from the CCS-like style, replacing parameter passing by direct assignments to state variables, and replacing gotos (encoded as tail-recursive process calls) by the higher-level control structures (loops, "**if-then-else**" conditionals, etc.) inherited from structured programming.

Fortunately, the LNT language is versatile enough to support both the CCS-like functional/recursive style and the imperative style with loops and variable assignments. The main reason for the functional style was to ensure that each variable is properly assigned before used (a prerequisite for having a formal semantics), but LNT resolves this issue in a much more flexible way, by using static analysis [10].

In version U3 of the Algorand model, two modules (`COUNTER.lnt` and `NODE.lnt`) are written in the CCS-like functional/recursive style.

Concerning `COUNTER.lnt`, the transformation to imperative style is straightforward (the corresponding code fragments can be found in Annexes B.5 and B.6, respectively). The three recursive calls in version U3 are replaced by one loop and assignments to variables in version U4. Both versions have

[10]The formal model given in [9, Sect. 3.3] manages to conceal verbosity by writing these parameters as tiny subscripts of processes N, N', ..., N'''' and by not declaring their types, letting readers infer unspecified types.

nearly the same number of LNT lines: deciding which one is easier to read is really a matter of individual taste. A minor advantage of version U4 is the strict encapsulation of both variables K_0 and K_1 in the *COUNTER* process, while version U3 exports these variables as parameters that need to be initialized by the caller process. This well-known issue with traditional process calculi could be resolved either by defining an additional process (without parameters) that calls the *COUNTER* process to initialize its K_0 and K_1 parameters, or by extending the LNT language with default values for parameters (as, e.g., in Ada or Python), which we prefer to avoid as it would interfere with overloading.

Concerning `NODE.lnt`, it is clear that the five mutually recursive processes N, N', ..., N'''' define one single state machine, since all process calls contained in them never return and rather express gotos between states than genuine process calls with a stack-based semantics. Thus, processes N, N', ..., N'''' must be considered and transformed altogether, as they form a unique piece of sequential code.

At first sight, the call graph of these processes, shown on Fig. 2 (left), is quite involved, so that using LNT loops instead of gotos is unlikely to increase conciseness and readability. But, a comparison with the algorithm of [7] suggests that the "true" states of the state machine may not merely be the five control locations N, N', ..., N''''. Instead, the "true" states seem to be pairs (L,S), where L is one of these five control locations and S is the current step, either 0, 1, or 2 (the latter value being also noted *S_INIT*).

A symbolic exploration of all reachable states (L,S) gives the call graph shown in Fig. 2 (middle), where (for conciseness of the figure) L_S denotes process L called with step parameter S, while processes N and N', which have no step parameter, are simply noted N and N'. Doing so, the inherent structure of the consensus algorithm appears, with an inner loop (in red) that cyclically executes the three "steps" defined in [7] (*Coin-Fixed-To-0*, *Coin-Fixed-To-1*, and *Coin-Genuinely-Flipped*), and an outer loop (in black) that contains the inner loop and performs an infinite sequence of "rounds", each of which receives and processes a new block.

Therefore, the consensus algorithm can easily be expressed using two nested LNT loops, the inner loop having three "**break**" statements that exit to the outer loop and correspond, in Fig. 2 (middle), to the three bottom-up edges displayed in black. Two of these edges go back to N, while the third edge goes back to N': this can easily be expressed in LNT by introducing a Boolean variable X that distinguishes between N and N' and is assigned before each "**break**" statement, together with an "**if-then**" conditional that only executes N if X has a given value, or jumps directly to N' otherwise.

Yet, one may notice that process N', which computes a random bit equal to zero with probability P, is called twice and plays a double role: (i) when called from process N with $P = P_H$, it models the result of Algorand's graded consensus phase (not modelled in detail) that takes place at the beginning of the outer loop, and (ii) when called from process N_2'''' with $P = 0.5$, it expresses the probabilistic choice of a random bit that takes place during the 3rd step (*Coin-Genuinely-Flipped*) of Algorand's consensus algorithm (see Annex B.7 for details). The factorization of (i) and (ii) in a single process N' was a design decision of the initial formal model [9] — which wrongly called N' with $P = P_H$ in the case (ii). Such a factorization somehow altered the structure of the algorithm by moving case (ii) before the inner loop, whereas case (ii) logically belongs to the inner loop, of which it is the last step.

Given that process N' is small, we decided to revert this factorization and have instead two copies of N', as it was originally the case in [7]. The resulting call graph is shown in Fig. 2 (right). The translation to LNT becomes simpler, as the inner loop now has only two "**break**" statements, and one no longer needs the aforementioned Boolean variable X nor the additional "**if-then**" conditional.

Consequently, the module `NODE.lnt` was fully rewritten in version U4 by removing the four processes N', ..., N''' after expanding their contents inline in process N, dispatching the various branches of the "**case**" statements where needed. The module `TYPES.lnt` was also simplified by removing the type *STEP* and two functions *NEXT* and *S_INIT*, which are no longer used in version U4. To factor out

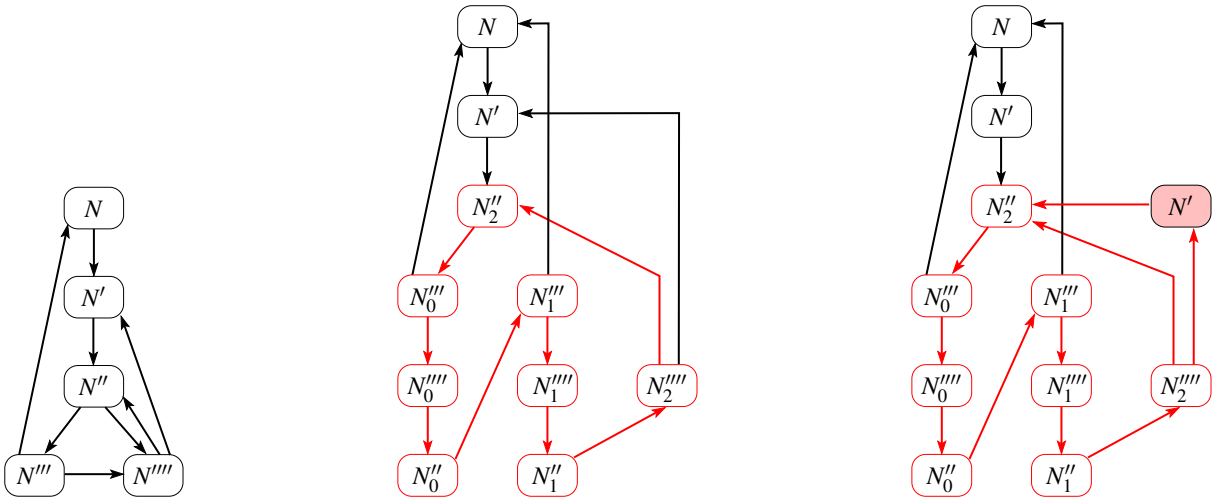


Figure 2: Loop discovery: call graph for processes only (left); call graph for pairs of processes and steps (middle); call graph for pairs processes and steps, with duplication of process N' (right)

duplicated code, the new process N can then be shortened by introducing three auxiliary processes that contain neither loops nor process calls:

- Process *FIX_COIN* encapsulates assignments to the bit B, noticing that each such assignment is always followed by a *SET_BIT* event.
- Process *FLIP_COIN* corresponds to the former (duplicated) process N' (i.e., the probabilistic computation of a random value for the bit B) at the end of which the initial *SET_BIT* event of process N'' was added.
- Process *BROADCAST* corresponds to the former process N'', from which the initial *SET_BIT* event was removed, and at the end of which the initial *TALLY* event of process N''' was added.

Thus, version U4 deeply changed the appearance of the model, making it look much closer to the algorithm of [7] and bringing new insight into Algorand. Compared to version U3, the 12 recursive process calls present in the `COUNTER.lnt` and `NODE.lnt` modules have been replaced by three LNT loops, and the number of lines of LNT code was further reduced by 13% (see Table in Sect. 2.2).

7 Formal Verification

As mentioned in Sect. 2.2, the LTSs corresponding to versions U0, ..., U4 are strongly bisimilar. Can one obtain further guarantees that these LNT models faithfully describe the Algorand consensus protocol?

A part of the answer lies the LNT compiler, which performs multiple checks based on control-flow and data-flow analyses. In earlier versions of the Algorand model, these checks detected various problems (e.g., wrong synchronizations likely to cause deadlocks), which have been fixed in [9] and U0.

Another part of the answer lies in state-space exploration methods (i.e., LTS construction) which, although expensive, are easily tractable for Algorand configurations with 4 nodes (see Table in Sect. 2.2). We now briefly present how such configurations can be analyzed using visual checking, equivalence checking, and model checking. Larger configurations have also been explored using the compositional verification [14] capabilities of CADP (e.g., 6 nodes in Grenoble, 8 and 10 nodes in Urbino).

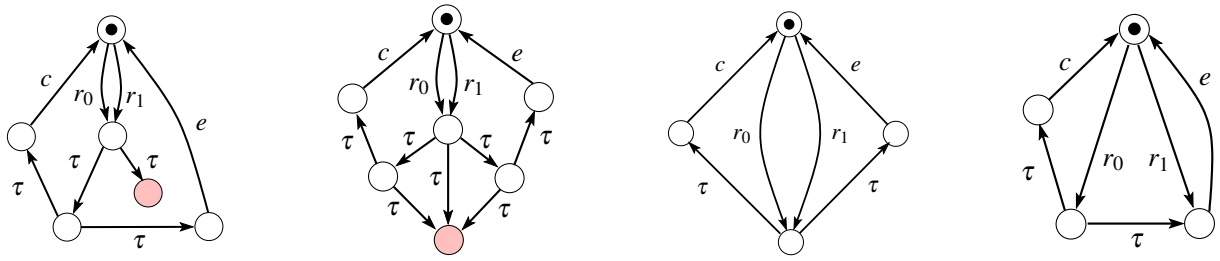


Figure 3: Visual checking – (a) and (b): deadlocks, (c): normal behaviour, (d): corrupted behaviour

7.1 Visual Checking

The LTSs generated for Algorand with 4 nodes have, after minimization for strong bisimulation, 12,000–20,000 states, 43,000–69,000 transitions, and 131 different labels. They are too complex for visual inspection by a human. One approach is to use "event slicing" abstractions, i.e., hiding (or renaming) all events but a few ones of interest, minimizing the LTS for branching bisimulation, and observing the minimized LTS, if it is small enough to be visually inspected. For instance, if only *SYNC* events are kept visible, the minimized LTS is an infinite loop of *SYNC (BEGIN)* events followed by *SYNC (END)* events, which gives a positive indication of correctness.

If only the four events corresponding to the upper-level interface of the consensus protocol are kept visible, namely, *RECEIVE_BLOCK_PROPOSAL (0 or 1)*, *COMMIT_PROPOSED_BLOCK*, and *COMMIT_EMPTY_BLOCK* (which we abbreviate as r_0, r_1, c, and e, respectively), other interesting properties can be discovered using visual checking. For instance, the presence of deadlocks in earlier versions of the protocol[11] can be seen easily (red states) in Fig. 3 (a) and (b) — the CADP tools then give the shortest sequences of events leading to these deadlock states. Also, by varying the number of honest nodes in 0...N and the threshold value T (used by the vote algorithm) in 1...N, one observes that, if $H \geq T$, the minimized LTS is that of Fig. 3 (c), which shows a normal behaviour where every block received (event r_0 or r_1) is either accepted (event c) or rejected[12] (event e). But, if $H < T$, i.e., if there are too many malicious nodes, the minimized LTS is that of Fig. 3 (d), in which attacks are clearly successful, since any block carrying the bit one received is systematically rejected. More examples of visually checked properties are given in Annex C.

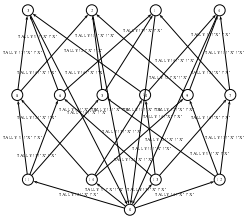
7.2 Equivalence Checking

Equivalence checking [12], as it is used in the present article, is a verification technique based on the comparison of two LTSs to decide whether they are bisimilar (e.g., for strong or branching bisimulation) or whether one LTS is included in the other for some behavioural preorder relation. As mentioned above, equivalence checking was already used to prove that the successive versions of the Algorand model are strongly bisimilar. But equivalence checking can be used more widely, e.g., to check whether a complex system (namely, an LTS generated from a formal model of Algorand, after hiding and/or renaming certain labels to keep only those events of interest) is branching bisimilar to a simpler system, the correctness of which is evident.

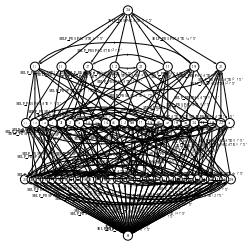
For instance, if one takes the (complex) LTS produced for a four-node Algorand configuration, hides

[11] i.e., prior to version v5 of the arXiv report [9].

[12] The rejection of a proposed block in Algorand is expressed by committing an empty block.



```
loop
 par
   tally (1,X,X)
   || ... ||
   tally (4,X,X)
 end par
end loop
```



```
loop
 par
   p(1) || ... || p(4)
 end par
end loop

where

process p(n) is
  τ;
  alt
    τ
  □
    τ;
    self_propagate(n,X)
  end alt
end process
```

Figure 4: Equivalence checking for *TALLY* events (left) and *SELF_PROPAGATE* events (right)

all events but *TALLY* (which is the event used by each node to query its counter about vote results), renames in each *TALLY (ID, K_0, K_1)* label the integer values K_0 and K_1 by some abstract constant X, and minimizes the result for branching bisimulation, one obtains the (simple) LTS shown in Fig. 4 (left). This LTS is a bit too large to be checked visually, but one easily guesses it is a cyclic four-dimensional hypercube that can be described by the LNT code fragment shown on its right. The CADP tools are then used to formally confirm this intuition.

Similarly, if one takes the same (complex) LTS, hides all events but *SELF_PROPAGATE* (which is the event used by each node to communicate its vote to its counter), renames in each *SELF_PROPAGATE (ID, B)* label the bit value *B* by some abstract constant X, and minimizes the result for branching bisimulation, one obtains the (not so simple) LTS shown in Fig. 4 (right). After many trials, it appears that this LTS can be expressed by the LNT code fragment shown on its right (with the definition of an auxiliary process *p*), which is confirmed by the CADP tools.

7.3 Model Checking

Visual checking and equivalence checking may be intractable if the model under verification is too large for bisimulation algorithms, or if the property to be verified is complex and cannot be easily expressed as a simple LTS. In both cases, model checking offers an alternative, in which the property is specified as a temporal-logic formula given to an algorithm that computes a Boolean truth value and a diagnostics (i.e., a counterexample) explaining why the formula is true or false on the model.

For instance, considering the *SET_BIT* event (which is the event introduced in the Algorand formal model to observe, at every step, how each node assigns its local bit *B*), event slicing for *SET_BIT (ID, STEP, B)* events does not give exploitable results. Even if one considers only one chosen value of *ID*, hiding all other labels and renaming all bit values *B* by some abstract constant X, the resulting LTSs minimized for branching bisimulation are small (25–50 states) but complex, with many (83%) hidden transitions (noted "τ" as in CCS [27]). It is thus easier to express the properties that *SET_BIT* events are expected to satisfy using temporal-logic formulas rather than behavioural relations. An example of such a formula, written in the MCL language [24] [25] and the SVL script language [11] of CADP, is shown in Fig. 5 — notice the value-passing capabilities of MCL, which are used to follow the successive values taken by variables *ID* and *STEP*. Other MCL properties are given in Annex C.

```
property P9b (MODEL)
 "for each ID in 1...4, after SET_BIT (ID, STEP, any), where STEP < 2, it is inevitable to reach"
 "either RECEIVE_BLOCK_PROPOSAL (any) or SET_BIT (ID, STEP + 1, 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 }) ;
 -- AFTER_1_WITHOUT_2_INEVITABLE_3 (x,y,z) is a predefined macro of the MCL standard library
end property
```

Figure 5: Model checking with a value-passing MCL formula embedded in an SVL script

8 Conclusion

Because they are designed to be minimal languages with tiny syntax and concise semantics, old-school process calculi often lead to poorly readable specifications — very much like low-level code written in assembly or bytecode languages. LNT supports the functional/recursive programming style of these process calculi, but also brings useful features (imperative programming style with direct assignments to variables, symmetric sequential composition, and explicit loop operators) that, although present in the original version of CSP [17], are missing in CCS and most process calculi inspired by CCS. Such a versatility in LNT allows models that are easier to read, maintain, and evolve. Thanks to these features, LNT bridges the gap between executable programs and formal models, by having a continuity in style between these two classes of artefacts.

To assess these claims, we experimented with a recent formal model [9] of the Algorand consensus protocol [7]. Starting from this initial version (967 lines of LNT), we applied specific transformations to produce a simpler version U0 (769 lines), which we further simplified, using four generic transformations, to obtain four successive versions U1, U2, U3, and U4. The latter version is concise (250 lines only), easily readable, and brings new insight into the protocol.

The four generic transformations presented in Sect. 3 to 6 are based on structural/algebraical properties of LNT and are thus applicable to other case studies. They can be completed with other generic transformations, such as those used in [16]: inline expansion of small auxiliary processes and flattening of nested "**if-then-else**" conditionals by adding "**elsif**" tests. All these transformations can be done in a stepwise manner, checking that strong or branching bisimulation is preserved at each step.

We also verified various properties using visual checking, equivalence checking, and model checking on the formal models of Algorand. The present work could be extended in at least four directions:

- Keeping the model as it is, one could devise more correctness properties to be verified. In particular, it would be challenging to specify MCL formulas that globally observe the votes of all nodes and check whether the decisions taken by consensus are really the expected ones.
- Because the model contains probabilistic choices, it would be natural to perform probabilistic verification. This would require to complete the model with the "graded consensus" phase (currently abstracted away) and to better describe the notion of committees, either by making sure that the size of committees is equal to C or, at least, by considering only committees whose size is larger than the threshold value T and does not have a too low occurrence probability.

- One could relax the synchrony assumption, materialized in the model by the two synchronizations barriers *SYNC (BEGIN)* and *SYNC (END)*, and adopt the "substantially asynchronous" setting of [7], in which timeouts and overlapping rounds must be taken into account.
- Finally, the model could shift from the BBA* protocol of [7] to its successor, the ABFT (Algorand Byzantine Fault Tolerance) protocol, which is implemented and deployed, but whose specifications [1] are dynamically evolving over time, possibly causing moving-target issues for modellers.

Acknowledgements

We are grateful to Marco Bernardo, Andrea Esposito, and Francesco Pio Rossi for undertaking a formal modelling of the Algorand consensus protocol and for their patient explanations of its intricacies. We also thank Pierre-Yves Piriou and the anonymous reviewers for their remarks about the present article. The development of versions U0–U4 led to more than 420 different models, which sometimes pushed the limits of LNT: acknowledgements are due to Frédéric Lang and Wendelin Serwe for promptly improving the LNT tools, and to Radu Mateescu for advising on the specification of MCL temporal-logic formulas.

References

[1] Algorand Foundation (2025): *Algorand Specifications*. Available at `https://specs.algorand.co`.

[2] J. A. Bergstra & J. W. Klop (1984): *Process Algebra for Synchronous Communication*. *Information and Computation* 60(1–3), pp. 109–137, doi:10.1016/S0019-9958(84)80025-X.

[3] Tommaso Bolognesi & Ed Brinksma (1988): *Introduction to the ISO Specification Language LOTOS*. *Computer Networks and ISDN Systems* 14(1), pp. 25–59, doi:10.1016/0169-7552(87)90085-7.

[4] E. Brinksma & G. Leih (1995): *Enhancements of LOTOS*. In T. Bolognesi, J. van de Lagemaat & C. Vissers, editors: *LOTOSphere: Software Development with LOTOS*, Kluwer Academic Publishers, pp. 453–466, doi:10.1007/978-1-4615-2203-4_22.

[5] Ed Brinksma (1988): *On the Design of Extended LOTOS – A Specification Language for Open Distributed Systems*. Ph.D. thesis, University of Twente.

[6] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Christine McKinty, Vincent Powazny, Frédéric Lang, Wendelin Serwe & Gideon Smeding (2026): *Reference Manual of the LNT to LOTOS Translator (Version 7.5)*. Available at `https://cadp.inria.fr/publications/Champelovier-Clerc-Garavel-et-al-10.html`. INRIA, Grenoble, France.

[7] Jing Chen & Silvio Micali (2019): *Algorand: A Secure and Efficient Distributed Ledger*. *Theoretical Computer Science* 777(2), pp. 155–183, doi:10.1016/j.tcs.2019.02.001.

[8] Edsger W. Dijkstra (1968): *Letters to the Editor: Go To Statement Considered Harmful*. *Communications of the ACM* 11(3), pp. 147–148, doi:10.1145/362929.362947.

[9] Andrea Esposito, Francesco P. Rossi, Marco Bernardo, Francesco Fabris & Hubert Garavel (2025): *Formal Modeling and Verification of the Algorand Consensus Protocol in CADP*. Technical Report arXiv:2508.19452, arXiv Computing Research Repository, doi:10.48550/arXiv.2508.19452.

[10] Hubert Garavel (2015): *Revisiting Sequential Composition in Process Calculi*. *Journal of Logical and Algebraic Methods in Programming* 84(6), pp. 742–762, doi:10.1016/j.jlamp.2015.08.001.

[11] Hubert Garavel & Frédéric Lang (2001): *SVL: a Scripting Language for Compositional Verification*. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang & Danhyung Lee, editors: *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'01), Cheju Island, Korea*, Kluwer Academic Publishers, pp. 377–392, doi:10.1007/0-306-47003-9_24. Full version available as INRIA Research Report RR-4223.

[12] Hubert Garavel & Frédéric Lang (2022): *Equivalence Checking 40 Years After: A Review of Bisimulation Tools*. In Nils Jansen, Marielle Stoelinga & Petra van den Bos, editors: *A Journey from Process Algebra via Timed Automata to Model Learning – Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*, *Lecture Notes in Computer Science* 13560, Springer, pp. 213–265, doi:10.1007/978-3-031-15629-8_13.

[13] Hubert Garavel, Frédéric Lang, Radu Mateescu & Wendelin Serwe (2013): *CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes*. *Springer International Journal on Software Tools for Technology Transfer (STTT)* 15(2), pp. 89–107, doi:10.1007/s10009-012-0244-z.

[14] Hubert Garavel, Frédéric Lang & Laurent Mounier (2018): *Compositional Verification in Action*. In Falk Howar & Jiri Barnat, editors: *Proceedings of the 23rd International Conference on Formal Methods for Industrial Critical Systems (FMICS'18), Maynooth, Ireland – Essays Dedicated to Susanne Graf at the Occasion of Her 60th Birthday*, *Lecture Notes in Computer Science* 11119, Springer, pp. 189–210, doi:10.1007/978-3-030-00244-2_13.

[15] Hubert Garavel, Frédéric Lang & Wendelin Serwe (2017): *From LOTOS to LNT*. In Joost-Pieter Katoen, Rom Langerak & Arend Rensink, editors: *ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, *Lecture Notes in Computer Science* 10500, Springer, pp. 3–26, doi:10.1007/978-3-319-68270-9_1.

[16] Hubert Garavel & Bas Luttik (2024): *Four Formal Models of IEEE 1394 Link Layer*. In Frédéric Lang & Matthias Volk, editors: *Proceedings of the 6th Workshop on Models for Formal Analysis of Real Systems (MARS'24), Luxembourg City, Luxembourg*, *EPTCS* 399, pp. 21–100, doi:10.4204/EPTCS.399.5.

[17] C. A. R. Hoare (1978): *Communicating Sequential Processes*. *Communications of the ACM* 21(8), pp. 666–677, doi:10.1145/359576.359585.

[18] C. A. R. Hoare (1991): *The Transputer and Occam: A Personal Story*. *Concurrency – Practice and Experience* 3(4), pp. 249–264, doi:10.1002/CPE.4330030403.

[19] C. Norris Ip & David L. Dill (1996): *Better Verification Through Symmetry*. *Formal Methods in System Design* 9, pp. 41–75, doi:10.1007/BF00625968.

[20] ISO/IEC (1989): *ESTELLE – A Formal Description Technique Based on an Extended State Transition Model*. International Standard 9074, International Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Geneva. Available at `https://www.iso.org/standard/16659.html`.

[21] ISO/IEC (1989): *LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. International Standard 8807, International Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Geneva. Available at `https://www.iso.org/standard/16258.html`.

[22] ISO/IEC (2001): *Enhancements to LOTOS (E-LOTOS)*. International Standard 15437:2001, International Organization for Standardization – Information Technology, Geneva. Available at `https://www.iso.org/standard/27680.html`.

[23] ITU-T (1999): *Specification and Description Language (SDL)*. ITU-T Recommendation Z.100, International Telecommunication Union, Geneva. Available at `https://www.itu.int/rec/T-REC-Z.100-199911-S`.

[24] Radu Mateescu (1998): *Vérification des propriétés temporelles des programmes parallèles*. Ph.D. thesis, Institut National Polytechnique de Grenoble. Available at `https://theses.hal.science/tel-00004896v1`.

[25] Radu Mateescu & Damien Thivolle (2008): *A Model Checking Language for Concurrent Value-Passing Systems*. In Jorge Cuellar, Tom Maibaum & Kaisa Sere, editors: *Proceedings of the 15th International Symposium on Formal Methods (FM'08), Turku, Finland*, *Lecture Notes in Computer Science* 5014, Springer, pp. 148–164, doi:10.1007/978-3-540-68237-0_12.

[26] David May (1983): *OCCAM*. *SIGPLAN Notices* 18(4), pp. 69–79, doi:10.1145/948176.948183.

[27] Robin Milner (1980): *A Calculus of Communicating Systems*. Lecture Notes in Computer Science 92, Springer, doi:10.1007/3-540-10235-3.

[28] Mihaela Sighireanu, Alban Catry, David Champelovier, Hubert Garavel, Frédéric Lang, Guillaume Schaeffer, Wendelin Serwe & Jan Stoecker (2025): *LNT User Manual (Version 3.17)*. INRIA/CONVECS, Grenoble, France, `https://vasy.inria.fr/ftp/traian/manual.pdf`, 92 pages.

[29] Mihaela Sighireanu & Hubert Garavel (1996): *On the Definition of Modular E-LOTOS*. VASY Report, INRIA. Available at `https://vasy.inria.fr/ftp/publications/elotos/elotos-grenoble-2.1.pdf`. Input Document [GR2] to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Grenoble, France, December 9–11, 1996.

A Preliminary Steps

A.1 Ambiguities in the Informal Description

We give two examples of ambiguities that we found in the original, informal description [7] and indicate which are the most plausible interpretations:

- The Algorand consensus protocol is temporally divided into successive "rounds", and each round is divided into successive "steps". One may wonder whether a committee is formed at each round, meaning that the committee remains the same while a given block is examined (interpretation #1) or at each step, meaning that several committees are formed for the same block (interpretation #2).

  The difference between rounds and steps is clearly stated in [7], e.g., page 160, where the variables r and s are defined as follows: "$r \geq 0$ *and* $s \geq 1$ *[denote] the current round and the current step (in a given round)*". In favor of interpretation #1, page 156 defines: "*a small set of selected verifiers, noted* SV^r*, referred to as the committee*", and this notation SV^r is used throughout pages 157–159, suggesting that the committee depends only on round r. In favor of interpretation #2, page 159 formulates the concept of "player replaceability" and pages 160-161 introduce a new notation $SV^{r,s}$ defined as "*the set of verifiers of step* s *of round* r.

  Following [9], all formal models in LNT adopt interpretation #2, meaning that a new committee is formed at each step of each round.

- Another ambiguity concerns the amount of information passed from a step to the next step. One may wonder whether this information is empty (interpretation #1) or not (interpretation #2).

  In favor of interpretation #1, page 158 states, when defining player replaceability: "*no internal states need to be maintained by a player from one step to another, and the protocol correctly reaches consensus even if each step is executed by a totally new (independently and randomly selected) set of players*". In favor of interpretation #2, pages 163–164 introduce, for each node i, a bit variable noted b_i, the value of which is passed from the current step to the next step.

  Following [9] again, all formal models in LNT adopt interpretation #2 and assume that each node manages a state variable *B:BIT* that persists during an entire round.

A.2 Simplifying Abstractions in the Formal Models

The formal model presented in [9], from which the LNT models are derived, notably simplifies the Algorand consensus protocol defined in [7] by making various abstractions intended to ease formal verification using finite-state methods. We briefly summarize the main abstractions:

- The number of nodes (i.e., users) in the network is fixed. Nodes cannot join or leave the blockchain dynamically.
- The proportion of honest vs malicious nodes is fixed. A node is either honest or malicious, and will remain so forever.
- Two particular configurations are studied: $A_{4,0}$ (a network with four honest nodes) and $A_{2,2}$ (a network with two honest nodes and two malicious nodes).
- Money and money transfers (e.g., rewards) are not modelled. All nodes are assumed to have the same stake.
- The model focuses on the second phase "BBA*" (generalized Boolean Byzantine Agreement) of the Algorand consensus protocol. The first phase "GC" (Graded Consensus) is not described and its result is abstracted away using a probabilistic choice.
- The role of leader nodes (elected during the GC phase) is not modelled.
- Cryptographic aspects (credentials, public keys, secret keys, signatures, etc.) are not described.
- Step numbers are abstracted away by using their residues modulo 3 (noted $\equiv 0$, $\equiv 1$, and $\equiv 2$).
- Probabilities are given constant values (e.g., 0.75 for the probability to be in the committee and 0.7424 for the probability that an elected leader is honest).
- The underlying network is assumed to be synchronous, whereas Algorand was designed for a more permissive setting — qualified as "highly asynchronous" or "substantially asynchronous" in [7]. The synchrony assumption greatly simplifies the formal model, since each voting phase is delimited by two synchronization barriers that concern all nodes in the network, so that delays and time bounds do not have to be formalized.
- The Algorand notion of committee is loosely modelled. Assuming that the network has N nodes, votes take place within committees of C nodes, where C is usually much smaller than N. But, since each committee is formed probabilistically (using verifiable random functions), its size may be different from C, as it determined by a binomial distribution centered at C. Because of this, the formal model does not only considers committees of size C, but all possible committees whose size ranges between 0 and N, which somehow undermines the concept of committee.

The formal model of [9] also introduces an attacker model (i.e., assumptions about malicious nodes) that can be summarized as follows:

- Malicious nodes cannot fork the blockchain (which is probabilistically impossible by design in Algorand) but they seek to disrupt it by preventing certain valid blocks from being committed.
- Malicious nodes influence the blockchain by setting their votes to bit one (which means rejection).
- Malicious nodes share a private network. At the beginning of each round, they may synchronize altogether on an event named *BOYCOTT* and decide to attack the proposed block.

A.3 Model-Specific Transformations

The initial version U0 was produced by starting from the LNT model presented in [9, Sect. 4.2 and 4.3], to which the following (syntactic and semantic) transformations were applied:

1. Modified spacing and indentation at various places.

2. Renamed process *C* to *COUNTER*.
3. Added "**ensure**" post-condition in function *N*.
4. Added constant function *C* (committee size).
5. Added constant function *H* (number of honest nodes).
6. Modified definition of function *T* (threshold of votes).
7. Added function *P_V* (probability that a node is selected).
8. Added function *P_H* (probability that an elected leader is honest).
9. Replaced variable *P_0* and its hard-coded value 0.7424 by *P_H*.
10. Replaced variable *IN* and its hard-coded value 0.75 by *P_V*.
11. Replaced "action" by "event" in comments. Shortened or simplified certain comments.
12. Added enumerated type *TAG* with two values (*BEGIN* and *END*) to distinguish between the two *SYNC* events. Updated channel *SYNCHRONIZE* accordingly.
13. Tagged all local events *ADJUST_BIT*, *ASK*, *COMPUTE_BIT*, *P_B*, *P_IN*, *P_OUT*, *REPLY*, *SELF_PROPAGATE*, and *SELF_VERIFY* by giving them a first offer *ID*, which is the number of the node that emits these events; these extra offers are needed to observe the system and express properties to be verified. Updated the corresponing channels *ASK*, *COMPUTE*, *PROBABILISTIC*, *REPLY*, *SELF_PROPAGATE*, and *VERIFY* accordingly.
14. Reduced the number of enumerated values in type *STEP* from four to three by merging *S_INIT* and *S_TWO*. Turned *S_INIT* into a constant function equal to *S_TWO*.
15. Renamed the three constructors *S_ZERO*, *S_ONE*, and *S_TWO* of type *STEP* to 0, 1, and 2 for conciseness and for handling them easily in MCL temporal-logic formulas.
16. Introduced a new type *PID* that replaces type *NAT* for node numbers; besides increased type safety, the definition of function *N* now derives from that of type *PID* and the "**where**" guard in process *COUNTER* becomes simpler (no need to check min-max bounds).
17. Replaced offers "0 **of** *BIT*" and "1 **of** *BIT*" by "0" and "1" in events *ASK* and *REPLY* because, as of CADP version 2025-k, the LNT2LOTOS translator now exploits channel definitions to resolve overloading.
18. Moved comments related to *SELF_PROPAGATE* and *PROPAGATE* events from channel definitions to the *COUNTER* process.
19. Added an "**only if**" guard in the *COUNTER* process to allow compositional state-space generation.
20. Moved *ADJUST_BIT* events, which were placed before bit assignments, after bit assignments.
21. Moved *COMPUTE_BIT* events, which were placed before bit assignments, after bit assignments.
22. Added a second offer *B:BIT* to both events *COMPUTE_BIT* and *ADJUST_BIT*, so as to observe bit values after assignments.
23. Introduced a new type *PROB*, distinct from reals, for probability values. Updated functions *P_H* and *P_V* accordingly.
24. Added a new function *1_MINUS* to compute probabilities. Simplified processes N' and N'' by using this new function.

25. Added a second parameter *P:PROB* to process N'. Modified process N'''' to invoke N' with probability value 0.5 (fair coin tossing) rather than *P_H*.
26. Split *P_B* events used to model probabilistic choices into two distinct events *P_ZERO* and *P_ONE*, since the introduction of probability 0.5 creates nondeterminism between both events *P_B (ID, P)* and *P_B (ID, 1_MINUS (P))*, making the model harder to observe and verify. The names *P_ZERO* and *P_ONE* follow Algorand's conventions, where consensus on bit zero (resp., one) means acceptance (resp., rejection) of the proposed block.
27. Shortened variable names *K_0* and *K_1* to *K0* and *K1*, respectively.
28. Merged both events *ASK* and *REPLY* into a single event *TALLY*, which simplifies the internal protocol used by each node to query its counter; from now on, each event *TALLY (ID, ?K0, ?K1)* replaces a former sequence of four events *ASK (ID, 0)* $\rightarrow$ *REPLY (ID, ?K0)* $\rightarrow$ *ASK (ID, 1)* $\rightarrow$ *REPLY (ID, ?K1)*, dividing by four the number of reachable states, as observed in experiments.
29. Removed *SELF_VERIFY* events, which bring no useful information, as every *SELF_VERIFY (ID)* is immediately followed by either a *P_IN (ID, ...)* or *P_OUT (ID, ...)* event, and reciprocally; moreover, the two counter variables *K0* and *K1* can be reset by *TALLY* events rather than *SELF_VERIFY* events; such removal reduces the number of reachable states by more than 25%. Removed the *VERIFY* channel too.
30. Merged both events *ADJUST_BIT (ID, B)* and *COMPUTE_BIT (ID, B)*, which now have roughly the same meaning, into a single event *SET_BIT (ID, B)*. Merged both channels *ADJUST* and *COMPUTE* into a single channel *SET_BIT*.
31. Inserted a second offer *S:STEP* in *SET_BIT (ID, S, B)* events, so as to observe the current step of the protocol every time a bit is assigned.
32. Revised the attacker model: formerly, malicious nodes had to synchronize altogether on the *BOYCOTT* event before attacking (i.e., tampering their votes to try rejecting the proposed block) and only attacked when this synchronization succeeded; also, from a verification point of view, if the *BOYCOTT* event was hidden, it was impossible to distinguish between attacks and non-attacks; now, malicious nodes no longer synchronize on *BOYCOTT* but attack if the contents of the proposed block matches a given pattern (which we abstract away as a bit value). Created a new channel *COMMIT* with an empty profile (no offers). Changed the channel of events *COMMIT_PROPOSED_BLOCK* and *COMMIT_EMPTY_BLOCK* from *BLOCK* to *COMMIT*. Added an offer *B:BIT* to event *RECEIVE_BLOCK_PROPOSAL* and channel *BLOCK*. Modified process *N* to trigger attacks when the proposed block carries bit one. Removed event *BOYCOTT* and channel *BOYCOTT*.

Finally, to obtain fair statistics about the transformations proposed in Sect. 4, we reverted, in the definition of process N'', some of the code simplifications presented in Sect. 4, which had been incorportated in [9] by anticipation.

B Formal Model in LNT

This annex gives the LNT code of versions U2, U3, and U4. We do not reproduce here versions U0 and U1, since their LNT code is nearly identical (plus duplicated definitions and minus "*M: MORALITY*" parameters) to that of version U2.

B.1 Type Definitions

This section reproduces the `TYPES.lnt` module of version U3, which defines various types and their related functions. The `TYPES.lnt` module of version U2 is identical to that of version U3, minus the "<>" function of type *STEP*. The `TYPES.lnt` module of version U4 is identical to that of version U3, minus the "=", "<>", *S_INIT*, and *NEXT* functions of type *STEP*.

```
module TYPES (BIT) is  -- the BIT type is imported from a predefined library

type PID is  -- node identifiers
   range 1...4 of NAT
   with <>, last
end type

type STEP is  -- steps of BBA* consensus protocol in [Chen-Micali-19]
   0,  -- congruent-to-0-modulo-3 step
   1,  -- congruent-to-1-modulo-3 step
   2   -- congruent-to-2-modulo-3 step
   with =, <>
end type

function S_INIT: STEP is  -- initialization step
   return 2
end function

function NEXT (S: STEP): STEP is  -- next step modulo 3
   case S in
      0 -> return 1
   |  1 -> return 2
   |  2 -> return 0
   end case
end function

type TAG is  -- synchronization tags
   BEGIN,
   END
end type

type MORALITY is  -- honesty values of nodes
   HONEST,
   MALICIOUS,
   DISGUISED  -- honest behavior of a malicious node
   with =
end type

type PROB is  -- probability
   P: REAL where 0.0 <= P and P <= 1.0
end type

function 1_MINUS (P: PROB): PROB is  -- complement of a probability
   return PROB (1.0 - REAL (P))
```

```
end function

end module
```

B.2 Constant Definitions

This subsection reproduces the `CONSTANT.lnt` module, which defines various constants and is identical in versions U1, U2, U3, and U4.

```
module CONSTANTS (TYPES) is

function N: NAT is   -- number of nodes in the network (derived from type PID)
    ensure result > 0;
    return NAT (last of PID)
end function

function C: NAT is   -- committee size (parameter)
    ensure 0 < result and result <= N;
    return 3
end function

function H: NAT is   -- number of honest nodes (parameter)
    ensure 0 <= result and result <= N;
    return 2   -- 2 for configuration A_(2,2), 4 for configuration A_(4,0)
end function

function T: NAT is   -- threshold of votes needed to commit a block (parameter)
    ensure result >= (2 * (C + 1)) div 3;   -- i.e., (2 / 3) * C, rounded up
    return 2
end function

function P_V: PROB is   -- probability that a node is selected for the committee
    return PROB (REAL (C) / REAL (N))
end function

function P_H: PROB is   -- probability that an elected leader is honest
    var P: REAL in
        P := REAL (H) / REAL (N);   -- probability that a node is honest
        return PROB ((P ^ 2.0) * (1.0 + P - (P ^ 2.0)))   -- from Chen-Micali-19
    end var
end function

end module
```

B.3 Channel Definitions

This subsection reproduces the `CHANNELS.lnt` module, which defines various channels (used to specify the types of LNT event parameters) and is identical in versions U2, U3, and U4.

```
module CHANNELS (TYPES) is

channel BLOCK is   -- for block proposals
   (V: BIT)
end channel

channel COMMIT is   -- for commit events
   ()
end channel

channel SYNCHRONIZE is   -- for synchronization events
   (T: TAG)
end channel

channel SET_BIT is   -- for bit setting events
   (ID: PID, S: STEP, B: BIT)
end channel

channel PROPAGATE is   -- for bit propagation events
   (ID: PID, B: BIT)
end channel

channel SELF_PROPAGATE is   -- for bit self-propagation events
   (ID: PID, B: BIT)
end channel

channel TALLY is   -- for tally events
   (ID: PID, K0: NAT, K1: NAT)
end channel

channel PROBABILISTIC is   -- for probabilistic choices
   (ID: PID, P: PROB)
end channel

end module
```

B.4 Main Process

This subsection reproduces the `ALGORAND.lnt` module, which defines an Algorand configuration with four nodes only and is identical in versions U1, U2, U3, and U4.

```
module ALGORAND (NODE) is

process MAIN [RECEIVE_BLOCK_PROPOSAL: BLOCK, COMMIT_PROPOSED_BLOCK: COMMIT,
              COMMIT_EMPTY_BLOCK: COMMIT, SYNC: SYNCHRONIZE, SET_BIT: SET_BIT,
              PROPAGATE: PROPAGATE, SELF_PROPAGATE: SELF_PROPAGATE,
              TALLY: TALLY, P_IN, P_OUT, P_ZERO, P_ONE: PROBABILISTIC] is
   par RECEIVE_BLOCK_PROPOSAL, PROPAGATE, SYNC,
       COMMIT_PROPOSED_BLOCK, COMMIT_EMPTY_BLOCK in
```

```
      NODE [...] (1 of PID)
   ||
      NODE [...] (2 of PID)
   ||
      NODE [...] (3 of PID)
   ||
      NODE [...] (4 of PID)
   end par
end process

end module
```

B.5 Counter Process (Version U3)

This subsection reproduces the `COUNTER.lnt` module that is specified in the CCS-like style (using only action prefix and tail process recursion) and is identical in versions U1, U2, and U3.

```
module COUNTER (TYPES, CONSTANTS, CHANNELS) is

process COUNTER [PROPAGATE: PROPAGATE, SELF_PROPAGATE: SELF_PROPAGATE,
                 TALLY: TALLY] (ID: PID, K0, K1: NAT) is
   var J: PID, B: BIT in
      alt
         only if K0 + K1 < N then
            alt
               -- propagation events go from other nodes to this counter
               PROPAGATE (?J, ?B) where J <> ID
            []
               -- self-propagation events go from each node to its counter
               SELF_PROPAGATE (ID, ?B)
            end alt;
            if B == 0 then
               COUNTER [...] (ID, K0 + 1, K1)
            else
               COUNTER [...] (ID, K0, K1 + 1)
            end if
         end if
      []
         TALLY (ID, K0, K1);
         COUNTER [...] (ID, 0, 0)
      end alt
   end var
end process

end module
```

B.6 Counter Process (Version U4)

This subsection reproduces the `COUNTER.lnt` module that is specified in the LNT imperative style (using assignments, symmetric sequential composition, and loop operators).

```
module COUNTER (TYPES, CONSTANTS, CHANNELS) is

process COUNTER [PROPAGATE: PROPAGATE, SELF_PROPAGATE: SELF_PROPAGATE,
                 TALLY: TALLY] (ID: PID) is
   var B: BIT, J: PID, K0, K1: NAT in
      K0 := 0;
      K1 := 0;
      loop
         alt
            only if K0 + K1 < N then
               alt
                  -- propagation events go from other nodes to this counter
                  PROPAGATE (?J, ?B) where J <> ID
               []
                  -- self-propagation events go from each node to its counter
                  SELF_PROPAGATE (ID, ?B)
               end alt;
               if B == 0 then
                  K0 += 1
               else
                  K1 += 1
               end if
            end if
         []
            TALLY (ID, K0, K1);
            K0 := 0;
            K1 := 0
         end alt
      end loop
   end var
end process

end module
```

B.7 Node Processes (Version U2)

This subsection reproduces the `NODE.lnt` module that is specified in the CCS-like style (using only action prefix and tail process recursion).

```
module NODE (TYPES, CONSTANTS, CHANNELS, COUNTER) is

process NODE [RECEIVE_BLOCK_PROPOSAL: BLOCK, COMMIT_PROPOSED_BLOCK: COMMIT,
              COMMIT_EMPTY_BLOCK: COMMIT, SYNC: SYNCHRONIZE, SET_BIT: SET_BIT,
              PROPAGATE: PROPAGATE, SELF_PROPAGATE: SELF_PROPAGATE,
```

```
6                TALLY: TALLY, P_IN, P_OUT, P_ZERO, P_ONE: PROBABILISTIC]
7               (ID: PID) is
8     par SELF_PROPAGATE, TALLY in
9        N [...] (ID)
10    ||
11       COUNTER [...] (ID, 0, 0)
12    end par
13 end process
14
15 process N [RECEIVE_BLOCK_PROPOSAL: BLOCK, COMMIT_PROPOSED_BLOCK: COMMIT,
16            COMMIT_EMPTY_BLOCK: COMMIT, SYNC: SYNCHRONIZE, SET_BIT: SET_BIT,
17            PROPAGATE: PROPAGATE, SELF_PROPAGATE: SELF_PROPAGATE,
18            TALLY: TALLY, P_IN, P_OUT, P_ZERO, P_ONE: PROBABILISTIC]
19           (ID: PID) is
20    var V: BIT in
21       RECEIVE_BLOCK_PROPOSAL (?V);
22       if NAT (ID) <= H then
23          N_PRIME [...] (ID, P_H, HONEST)
24       elsif V = 1 then
25          N_PRIME [...] (ID, P_H, MALICIOUS)
26       else
27          N_PRIME [...] (ID, P_H, DISGUISED)
28       end if
29    end var
30 end process
31
32 process N_PRIME [RECEIVE_BLOCK_PROPOSAL: BLOCK, COMMIT_PROPOSED_BLOCK: COMMIT,
33                  COMMIT_EMPTY_BLOCK: COMMIT, SYNC: SYNCHRONIZE, SET_BIT: SET_BIT,
34                  PROPAGATE: PROPAGATE, SELF_PROPAGATE: SELF_PROPAGATE,
35                  TALLY: TALLY, P_IN, P_OUT, P_ZERO, P_ONE: PROBABILISTIC]
36                 (ID: PID, P: PROB, M: MORALITY) is
37    alt
38       P_ZERO (ID, P);
39       N_SECOND [...] (ID, S_INIT, 0, M)
40    []
41       P_ONE (ID, 1_MINUS (P));
42       N_SECOND [...] (ID, S_INIT, 1, M)
43    end alt
44 end process
45
46 process N_SECOND [RECEIVE_BLOCK_PROPOSAL: BLOCK, COMMIT_PROPOSED_BLOCK: COMMIT,
47                   COMMIT_EMPTY_BLOCK: COMMIT, SYNC: SYNCHRONIZE, SET_BIT: SET_BIT,
48                   PROPAGATE: PROPAGATE, SELF_PROPAGATE: SELF_PROPAGATE,
49                   TALLY: TALLY, P_IN, P_OUT, P_ZERO, P_ONE: PROBABILISTIC]
50                  (ID: PID, S: STEP, in var B: BIT, M: MORALITY) is
51    SET_BIT (ID, NEXT (S), B);
52    B := B or BIT (M = MALICIOUS);
53    SYNC (BEGIN);
54    case S in
55       (* S_INIT *) 2 ->     -- the LNT compiler wants a constructor here
56          alt
57             P_IN (ID, P_V);
```

```
58              PROPAGATE (ID, B);
59              SELF_PROPAGATE (ID, B);
60              SYNC (END);
61              N_THIRD [...] (ID, 0, M)
62           []
63              P_OUT (ID, 1_MINUS (P_V));
64              SYNC (END);
65              N_THIRD [...] (ID, 0, M)
66           end alt
67      | 0 ->
68           alt
69              P_IN (ID, P_V);
70              PROPAGATE (ID, B);
71              SELF_PROPAGATE (ID, B);
72              SYNC (END);
73              N_THIRD [...] (ID, 1, M)
74           []
75              P_OUT (ID, 1_MINUS (P_V));
76              SYNC (END);
77              N_THIRD [...] (ID, 1, M)
78           end alt
79      | 1 ->
80           var K0, K1: NAT in
81              alt
82                 P_IN (ID, P_V);
83                 PROPAGATE (ID, B);
84                 SELF_PROPAGATE (ID, B);
85                 SYNC (END);
86                 TALLY (ID, ?K0 where K0 <= N, ?K1 where K1 <= N);
87                 N_FOURTH [...] (ID, 2, K0, K1, M)
88              []
89                 P_OUT (ID, 1_MINUS (P_V));
90                 SYNC (END);
91                 TALLY (ID, ?K0 where K0 <= N, ?K1 where K1 <= N);
92                 N_FOURTH [...] (ID, 2, K0, K1, M)
93              end alt
94           end var
95      end case
96   end process
97
98   process N_THIRD [RECEIVE_BLOCK_PROPOSAL: BLOCK, COMMIT_PROPOSED_BLOCK: COMMIT,
99                    COMMIT_EMPTY_BLOCK: COMMIT, SYNC: SYNCHRONIZE, SET_BIT: SET_BIT,
100                   PROPAGATE: PROPAGATE, SELF_PROPAGATE: SELF_PROPAGATE,
101                   TALLY: TALLY, P_IN, P_OUT, P_ZERO, P_ONE: PROBABILISTIC]
102                  (ID: PID, S: STEP, M: MORALITY) is
103      require S = 0 or S = 1;
104      var K0, K1: NAT in
105         case S in
106            0 ->
107               TALLY (ID, ?K0 where K0 <= N, ?K1 where K1 <= N);
108               if K0 >= T then
109                  COMMIT_PROPOSED_BLOCK;
```

```
110                N [...] (ID)
111             else
112                N_FOURTH [...] (ID, 0, K0, K1, M)
113             end if
114       |  1 ->
115             TALLY (ID, ?K0 where K0 <= N, ?K1 where K1 <= N);
116             if K1 >= T then
117                COMMIT_EMPTY_BLOCK;
118                N [...] (ID)
119             else
120                N_FOURTH [...] (ID, 1, K0, K1, M)
121             end if
122       |  any ->
123             raise UNEXPECTED
124       end case
125    end var
126 end process
127
128 process N_FOURTH [RECEIVE_BLOCK_PROPOSAL: BLOCK, COMMIT_PROPOSED_BLOCK: COMMIT,
129                   COMMIT_EMPTY_BLOCK: COMMIT, SYNC: SYNCHRONIZE, SET_BIT: SET_BIT,
130                   PROPAGATE: PROPAGATE, SELF_PROPAGATE: SELF_PROPAGATE,
131                   TALLY: TALLY, P_IN, P_OUT, P_ZERO, P_ONE: PROBABILISTIC]
132                  (ID: PID, S: STEP, K0, K1: NAT, M: MORALITY) is
133    case S in
134       0 ->
135          if K1 >= T then
136             N_SECOND [...] (ID, 0, 1, M)
137          else
138             N_SECOND [...] (ID, 0, 0, M)
139          end if
140    |  1 ->
141          if K0 >= T then
142             N_SECOND [...] (ID, 1, 0, M)
143          else
144             N_SECOND [...] (ID, 1, 1, M)
145          end if
146    |  2 ->
147          if K0 >= T then
148             N_SECOND [...] (ID, S_INIT, 0, M)
149          elsif K1 >= T then
150             N_SECOND [...] (ID, S_INIT, 1, M)
151          else
152             N_PRIME [...] (ID, 0.5 of PROB, M)
153          end if
154    end case
155 end process
156
157 end module
```

B.8 Node Processes (Version U3)

This subsection reproduces the `NODE.lnt` module that is specified in an halfway style combining elements from the CCS-like style (tail process recursion) and the LNT imperative style (assignments and symmetric sequential composition).

```
module NODE (TYPES, CONSTANTS, CHANNELS, COUNTER) is

process NODE [RECEIVE_BLOCK_PROPOSAL: BLOCK, COMMIT_PROPOSED_BLOCK: COMMIT,
              COMMIT_EMPTY_BLOCK: COMMIT, 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) is
   par SELF_PROPAGATE, TALLY in
      N [...] (ID)
   ||
      COUNTER [...] (ID, 0, 0)
   end par
end process

process N [RECEIVE_BLOCK_PROPOSAL: BLOCK, COMMIT_PROPOSED_BLOCK: COMMIT,
           COMMIT_EMPTY_BLOCK: COMMIT, 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) 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

process N_PRIME [RECEIVE_BLOCK_PROPOSAL: BLOCK, COMMIT_PROPOSED_BLOCK: COMMIT,
                 COMMIT_EMPTY_BLOCK: COMMIT, 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, P: PROB, M: MORALITY) is
   var B: BIT in
      alt
         P_ZERO (ID, P);
         B := 0
      []
         P_ONE (ID, 1_MINUS (P));
         B := 1
      end alt;
```

```
46        N_SECOND [...] (ID, S_INIT, B, M)
47     end var
48  end process
49
50  process N_SECOND [RECEIVE_BLOCK_PROPOSAL: BLOCK, COMMIT_PROPOSED_BLOCK: COMMIT,
51                    COMMIT_EMPTY_BLOCK: COMMIT, SYNC: SYNCHRONIZE, SET_BIT: SET_BIT,
52                    PROPAGATE: PROPAGATE, SELF_PROPAGATE: SELF_PROPAGATE,
53                    TALLY: TALLY, P_IN, P_OUT, P_ZERO, P_ONE: PROBABILISTIC]
54                   (ID: PID, S: STEP, in var B: BIT, M: MORALITY) is
55     SET_BIT (ID, NEXT (S), B);
56     B := B or BIT (M = MALICIOUS);
57     SYNC (BEGIN);
58     alt
59        P_IN (ID, P_V);
60        PROPAGATE (ID, B);
61        SELF_PROPAGATE (ID, B)
62     []
63        P_OUT (ID, 1_MINUS (P_V))
64     end alt;
65     SYNC (END);
66     case S in
67        (* S_INIT *) 2 ->     -- the LNT compiler wants a constructor here
68           N_THIRD [...] (ID, 0, M)
69     |  0 ->
70           N_THIRD [...] (ID, 1, M)
71     |  1 ->
72           var K0, K1: NAT in
73              TALLY (ID, ?K0 where K0 <= N, ?K1 where K1 <= N);
74              N_FOURTH [...] (ID, S_INIT, K0, K1, M)
75           end var
76     end case
77  end process
78
79  process N_THIRD [RECEIVE_BLOCK_PROPOSAL: BLOCK, COMMIT_PROPOSED_BLOCK: COMMIT,
80                   COMMIT_EMPTY_BLOCK: COMMIT, SYNC: SYNCHRONIZE, SET_BIT: SET_BIT,
81                   PROPAGATE: PROPAGATE, SELF_PROPAGATE: SELF_PROPAGATE,
82                   TALLY: TALLY, P_IN, P_OUT, P_ZERO, P_ONE: PROBABILISTIC]
83                  (ID: PID, S: STEP, M: MORALITY) is
84     require S = 0 or S = 1;
85     var K0, K1: NAT in
86        TALLY (ID, ?K0 where K0 <= N, ?K1 where K1 <= N);
87        if (S = 0 and K0 >= T) or (S = 1 and K1 >= T) then
88           if S = 0 then
89              COMMIT_PROPOSED_BLOCK
90           else
91              COMMIT_EMPTY_BLOCK
92           end if;
93           N [...] (ID)
94        else
95           N_FOURTH [...] (ID, S, K0, K1, M)
96        end if
97     end var
```

```
end process

process N_FOURTH [RECEIVE_BLOCK_PROPOSAL: BLOCK, COMMIT_PROPOSED_BLOCK: COMMIT,
                  COMMIT_EMPTY_BLOCK: COMMIT, 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, S: STEP, K0, K1: NAT, M: MORALITY) is
   var DONE: BOOL, B: BIT in
      DONE := false;
      B := 0;   -- dummy assignment to please the LNT compiler
      if S <> 0 then
         DONE := K0 >= T or S = 1;
         B := not (BIT (K0 >= T))
      end if;
      if not (DONE) and S <> 1 then
         DONE := K1 >= T or S = 0;
         B := BIT (K1 >= T)
      end if;
      if DONE then
         N_SECOND [...] (ID, S, B, M)
      else
         assert S = 2;
         N_PRIME [...] (ID, 0.5 of PROB, M)
      end if
   end var
end process

end module
```

B.9 Node Processes (Version U4)

This subsection reproduces the `NODE.lnt` module that is specified in the LNT imperative style (using assignments, symmetric sequential composition, and loop operators).

```
module NODE (TYPES, CONSTANTS, CHANNELS, COUNTER) is

process NODE [RECEIVE_BLOCK_PROPOSAL: BLOCK, COMMIT_PROPOSED_BLOCK: COMMIT,
              COMMIT_EMPTY_BLOCK: COMMIT, 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) is
   par SELF_PROPAGATE, TALLY in
      N [...] (ID)
   ||
      COUNTER [...] (ID)
   end par
end process

process N [RECEIVE_BLOCK_PROPOSAL: BLOCK, COMMIT_PROPOSED_BLOCK: COMMIT,
```

```
16            COMMIT_EMPTY_BLOCK: COMMIT, SYNC: SYNCHRONIZE, SET_BIT: SET_BIT,
17            PROPAGATE: PROPAGATE, SELF_PROPAGATE: SELF_PROPAGATE,
18            TALLY: TALLY, P_IN, P_OUT, P_ZERO, P_ONE: PROBABILISTIC]
19           (ID: PID) is
20   var B, V: BIT, K0, K1: NAT, M: MORALITY in
21      loop
22         RECEIVE_BLOCK_PROPOSAL (?V);
23         if NAT (ID) <= H then
24            M := HONEST
25         elsif V = 1 then
26            M := MALICIOUS
27         else
28            M := DISGUISED
29         end if;
30         FLIP_COIN [...] (ID, 0, ?B, P_H);  -- result of graded consensus phase
31         loop L in
32            -- Coin-Fixed-To-0 step
33            BROADCAST [...] (ID, B, M, ?K0, ?K1);
34            if K0 >= T then
35               COMMIT_PROPOSED_BLOCK;
36               break L
37            end if;
38            FIX_COIN [...] (ID, 1, ?B, BIT (K1 >= T));
39            -- Coin-Fixed-To-1 step
40            BROADCAST [...] (ID, B, M, ?K0, ?K1);
41            if K1 >= T then
42               COMMIT_EMPTY_BLOCK;
43               break L
44            end if;
45            FIX_COIN [...] (ID, 2, ?B, not (BIT (K0 >= T)));
46            -- Coin-Genuinely-Flipped step
47            BROADCAST [...] (ID, B, M, ?K0, ?K1);
48            if K0 >= T then
49               FIX_COIN [...] (ID, 0, ?B, 0)
50            elsif K1 >= T then
51               FIX_COIN [...] (ID, 0, ?B, 1)
52            else
53               FLIP_COIN [...] (ID, 0, ?B, 0.5 of PROB)  -- fair coin tossing
54            end if
55         end loop
56      end loop
57   end var
58 end process
59
60 process FIX_COIN [SET_BIT: SET_BIT]
61                  (ID: PID, S: STEP, out var B: BIT, NEW_B: BIT) is
62    B := NEW_B;
63    SET_BIT (ID, S, B)
64 end process
65
66 process FLIP_COIN [SET_BIT: SET_BIT, P_ZERO, P_ONE: PROBABILISTIC]
67                   (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 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

end module
```

C Verified Properties

The present annex lists a few properties, expressed in the SVL language[13] [11], which have been verified using CADP on both configurations $A_{4,0}$ and $A_{2,2}$ of version U4.

C.1 Absence of Deadlocks

```
property P1 (MODEL)
   "there is no deadlock in $MODEL.bcg"
is
   deadlock of "$MODEL.bcg";
   expected FALSE
end property
```

C.2 Properties of SYNC Events

[13] https://cadp.inria.fr/man/svl.html

```
property P2 (MODEL)
   "the events SYNC (BEGIN) and SYNC (END) alternate forever in $MODEL.bcg"
is
   branching comparison
      hide all but SYNC in "$MODEL.bcg" == "SYNC.lnt";
   expected TRUE
end property
```

where the auxiliary file `SYNC.lnt` contains the following definition:

```
module SYNC (TYPES, CHANNELS) is

process MAIN [SYNC: SYNCHRONIZE] is
   loop
      SYNC (BEGIN);
      SYNC (END)
   end loop
end process

end module
```

C.3 Properties of TALLY Events

```
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
```

```
property P3b (MODEL)
   "the events TALLY (ID, X, X) form a cyclic 4-dimensional hypercube"
is
   branching comparison
    total rename "TALLY !\([0-9]\) !\([0-9]\) !\([0-9]\)" ->  "TALLY !\1 !\"X\" !\"X\"" in
         hide all but TALLY in
            "$MODEL.bcg"
   ==
     "TALLY.lnt";
   expected TRUE
end property
```

where the auxiliary file `TALLY.lnt` contains the following definition:

```
module TALLY (TYPES) is

process MAIN [TALLY: any] is
   loop
      par
```

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

end module
```

C.4 Properties of PROPAGATE Events

```
property P4a (MODEL)
   "the events PROPAGATE (1, any) form a simple looping automaton"
is
   branching comparison
      total hide all but "PROPAGATE !1 !.*" in "$MODEL.bcg"
   ==
      generation of "PROPAGATE_1.lnt";
   expected TRUE
end property
```

where the auxiliary file `PROPAGATE_1.lnt` contains the following definition:

```
module PROPAGATE_1 (TYPES, CHANNELS) is

process MAIN [PROPAGATE: PROPAGATE] is
   loop
      alt
         i;
         PROPAGATE (1 of PID, 0 of BIT)
      []
         i;
         PROPAGATE (1 of PID, 1 of BIT)
      end alt
   end loop
end process

end module
```

```
property P4b (MODEL)
   "the events PROPAGATE (any, X) form a complex looping automaton"
is
   branching comparison
      total rename "PROPAGATE !\([0-9]\) !.*" -> "PROPAGATE !\1 !\"X\"" in
         hide all but PROPAGATE in
            "$MODEL.bcg"
   ==
```

```
        generation of "PROPAGATE.lnt";
    expected TRUE
end property
```

where the auxiliary file `PROPAGATE.lnt` contains the following definition:

```
module PROPAGATE (TYPES) is

process PART [PROPAGATE: any] (ID: PID) is
    alt
        i; PROPAGATE (ID, "X")
    []
        i
    end alt
end process

process MAIN [PROPAGATE: any] is
    loop
        par
            PART [...] (1 of PID)
        ||
            PART [...] (2 of PID)
        ||
            PART [...] (3 of PID)
        ||
            PART [...] (4 of PID)
        end par
    end loop
end process

end module
```

C.5 Properties of SELF_PROPAGATE Events

```
property P5a (MODEL)
    "the events SELF_PROPAGATE (1, any) form a simple looping automaton"
is
    branching comparison
        total hide all but "SELF_PROPAGATE !1 !.*" in "$MODEL.bcg"
    ==
        generation of "SELF_PROPAGATE_1.lnt";
    expected TRUE
end property
```

where the auxiliary file `SELF_PROPAGATE_1.lnt` contains the following definition:

```
module SELF_PROPAGATE_1 (TYPES, CHANNELS) is

process MAIN [SELF_PROPAGATE: SELF_PROPAGATE] is
    loop
        var B: BIT in
```

```
        B := any BIT;
        i;
        SELF_PROPAGATE (1 of PID, B)
      end var
    end loop
end process

end module
```

```
property P5b (MODEL)
   "the events SELF_PROPAGATE (any, X) form a complex looping automaton"
is
   branching comparison
      total rename "SELF_PROPAGATE !\([0-9]\) !.*" -> "SELF_PROPAGATE !\1 !\"X\"" in
         hide all but SELF_PROPAGATE in
            "$MODEL.bcg"
   ==
      generation of "SELF_PROPAGATE.lnt";
   expected TRUE
end property
```

where the auxiliary file `SELF_PROPAGATE.lnt` contains the following definition:

```
module SELF_PROPAGATE (TYPES) is

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

process MAIN [SELF_PROPAGATE: any] is
   loop
      par
         PART [...] (1 of PID)
      ||
         PART [...] (2 of PID)
      ||
         PART [...] (3 of PID)
      ||
         PART [...] (4 of PID)
      end par
   end loop
end process

end module
```

C.6 Properties of P_IN and P_OUT Events

```
property P6a (MODEL)
   "the events P_IN (1, any) and P_OUT (1, any) form a simple looping automaton"
is
   branching comparison
      total hide all but "P_IN !1 !.*", "P_OUT !1 !.*" in "$MODEL.bcg"
   ==
      generation of "IN_OUT_1.lnt";
   expected TRUE
end property
```

where the auxiliary file `IN_OUT_1.lnt` contains the following definition:

```
module IN_OUT_1 (TYPES, CHANNELS, CONSTANTS) is

process MAIN [P_IN, P_OUT: PROBABILISTIC] is
   loop
      alt
         P_IN (1 of PID, P_V)
      []
         P_OUT (1 of PID, 1_MINUS (P_V))
      end alt
   end loop
end process

end module
```

```
property P6b (MODEL)
   "the events P_IN and P_OUT form a complex looping automaton"
is
   branching comparison
      hide all but P_IN, P_OUT in "$MODEL.bcg" == generation of "IN_OUT.lnt";
   expected TRUE
end property
```

where the auxiliary file `IN_OUT.lnt` contains the following definition:

```
module IN_OUT (TYPES, CHANNELS, CONSTANTS) is

process ALT [P_IN, P_OUT: PROBABILISTIC] (ID: PID) is
   alt
      P_IN (ID, P_V)
   []
      P_OUT (ID, 1_MINUS (P_V))
   end alt
end process

process MAIN [P_IN, P_OUT: PROBABILISTIC] is
   loop
      par
         ALT [...] (1 of PID)
```

```
      ||
         ALT [...] (2 of PID)
      ||
         ALT [...] (3 of PID)
      ||
         ALT [...] (4 of PID)
      end par
   end loop
end process

end module
```

C.7 Properties of P_ZERO and P_ONE Events

```
property P7a (MODEL, H)   -- H: number of honest nodes in the configuration
   "the events P_ZERO (1, any) and P_ONE (1, any) form a simple looping automaton"
is
   branching comparison
      total hide all but "P_ZERO !1 !.*", "P_ONE !1 !.*" in "$MODEL.bcg"
   ==
      generation of "ZERO_ONE_1.lnt":MAIN [...] ($H);
   expected TRUE
end property
```

where the auxiliary file `ZERO_ONE_1.lnt` contains the following definition:

```
module ZERO_ONE_1 (TYPES, CHANNELS, CONSTANTS) is

process ALT [P_ZERO, P_ONE: PROBABILISTIC] (P: PROB) is
   alt
      P_ZERO (1 of PID, P)
   []
      P_ONE (1 of PID, 1_MINUS (P))
   end alt
end process

process MAIN [P_ZERO, P_ONE: PROBABILISTIC] (H: NAT) is
   var P: REAL, P_H: PROB in
      P := REAL (H) / REAL (N);
      P_H := PROB ((P ^ 2.0) * (1.0 + P - (P ^ 2.0)));
      ALT [...] (P_H);
      loop
         alt
            i;
            ALT [...] (0.5 of PROB)
         []
            i;
            ALT [...] (P_H)
         end alt
      end loop
   end var
```

```
end process

end module
```

```
property P7b (MODEL, H)  -- H: number of honest nodes in the configuration
   "the events P_ZERO and P_ONE form a complex looping automaton"
is
   branching comparison
      hide all but P_ZERO, P_ONE in "$MODEL.bcg"
   ==
      generation of "ZERO_ONE.lnt":MAIN [...] ($H);
   expected TRUE
end property
```

where the auxiliary file `ZERO_ONE.lnt` contains the following definition:

```
module ZERO_ONE (TYPES, CHANNELS, CONSTANTS) is

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

process PART [P_ZERO, P_ONE: PROBABILISTIC] (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 MAIN [P_ZERO, P_ONE: PROBABILISTIC] (H: NAT) is
   var P: REAL, P_H: PROB in
      P := REAL (H) / REAL (N);
      P_H := PROB ((P ^ 2.0) * (1.0 + P - (P ^ 2.0)));
      PART [...] (P_H);
      loop
         alt
            i;
            PART [...] (0.5 of PROB)
         []
            i;
            PART [...] (P_H)
         end alt
      end loop
```

```
   end var
end process

end module
```

C.8 Properties of BLOCK-related Events

```
property P8a (MODEL)
   "observed alone, the events RECEIVE_BLOCK_PROPOSAL, COMMIT_PROPOSED_BLOCK,"
   "and COMMIT_EMPTY_BLOCK show the absence or presence of successful attacks"
is
   "${MODEL}_min.bcg" = hide all but RECEIVE_BLOCK_PROPOSAL,
                        COMMIT_PROPOSED_BLOCK, COMMIT_EMPTY_BLOCK in
                          "$MODEL.bcg";
   branching comparison "${MODEL}_min.bcg" == generation of "OK.lnt";
   result R1;
%   if [ $R1 = "TRUE" ]
%   then
%      echo "OK"
%      return
%   fi
   branching comparison "${MODEL}_min.bcg" == generation of "KO.lnt";
   result R2;
%   if [ $R2 = "TRUE" ]
%   then
%      echo "KO"
%      return
%   fi
end property
```

where the auxiliary file `OK.lnt` contains the following definition:

```
module OK (TYPES, CHANNELS) is

process MAIN [RECEIVE_BLOCK_PROPOSAL: BLOCK, COMMIT_PROPOSED_BLOCK: COMMIT,
              COMMIT_EMPTY_BLOCK: COMMIT] is
   loop
      RECEIVE_BLOCK_PROPOSAL (?any BIT);
      alt
         i;
         COMMIT_PROPOSED_BLOCK
      []
         i;
         COMMIT_EMPTY_BLOCK
      end alt
   end loop
end process

end module
```

and the auxiliary file `KO.lnt` contains the following definition:

```
module KO (TYPES, CHANNELS) is

process MAIN [RECEIVE_BLOCK_PROPOSAL: BLOCK, COMMIT_PROPOSED_BLOCK: COMMIT,
              COMMIT_EMPTY_BLOCK: COMMIT] is
   loop
      alt
         RECEIVE_BLOCK_PROPOSAL (0 of BIT);
         alt
            i;
            COMMIT_PROPOSED_BLOCK
         []
            i;
            COMMIT_EMPTY_BLOCK
         end alt
      []
         RECEIVE_BLOCK_PROPOSAL (1 of BIT);
         COMMIT_EMPTY_BLOCK
      end alt
   end loop
end process

end module
```

C.9 Properties of SET_BIT Events

```
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
```

```
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
```

```
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
```