Full Proof of Reduction Techniques Employed for Model Checking Concurrency in the Tendermint Byzantine Fault Tolerant Consensus Protocol

1 Full Model Code

A full version of the model code is provided here for easy reference.

```
#import "PAT.Lib.Tendermint_v4";


#define N 4;
#define F 1;
#define INIT_TIMEOUT_PROPOSE 3;
#define INIT_TIMEOUT_PREVOTE 3;
#define INIT_TIMEOUT_PRECOMMIT 3;
#define TIMEOUT_DELTA 1;
#define BOUND_DELTA 4;
#define MESSAGE_LOG_DEFAULT_VALUE -2;

enum {UNDEFINED, PROPOSE, PREVOTE, PRECOMMIT};
enum {NIL, DECISION_T, DECISION_F};

// 3-bit flag where
// the first one is for UponSufficientPrevoteAny, and
// the second one is for UponSufficientPrevoteValue, and
// the third one is for UponSufficientPrecommitAny
var effective_rounds = [0(N)];
var rounds:{0..2} = [0(N)];
var steps: {0..3} = [UNDEFINED(N)];
var decisions: {0..2} = [NIL(N)];
var locked_values: {0..2} = [NIL(N)];
var locked_rounds: {-1..2} = [-1(N)];
var valid_values: {0..2} = [NIL(N)];
var valid_rounds: {-1..2} = [-1(N)];
var proposals: {0..2} = [NIL(N)];
var<MessageLog> message_log = new MessageLog();
hvar<AllMessages> all_messages = new AllMessages();
hvar honest_processes = [true, true, true, false];


BroadcastProposalMessage(p, round, proposal, valid_round)
    =
```

```
    broadcast_proposal_message_to_all_processes_from_process.p{
     message_log.AddProposal(new ProposalMessage(p, round, proposal, valid_round));
   } ->
   Skip;

BroadcastPrevoteMessage(p, round, prevote) =
   broadcast_prevote_message_to_all_processes_from_process.p{
     message_log.AddPrevote(new PrevoteMessage(p, round, prevote));
   } ->
   Skip;

BroadcastPrecommitMessage(p, round, precommit) =

    broadcast_precommit_message_to_all_processes_from_process.p{
     message_log.AddPrecommit(new PrecommitMessage(p, round, precommit));
   } ->
   Skip;

StartRound(p, round) =
   ifa (round >= rounds[p]) {
     reset_all_flags_in_effective_rounds_for_process.p{
       effective_rounds[p] = 0;
     } ->
     update_round_and_step_for_process_in_round.p.round{
       rounds[p] = round;
       steps[p] = PROPOSE;
     } ->
     ifa (round % N == p) {
       ifa (valid_values[p] != NIL) {

    update_proposal_to_valid_value_for_process_in_round.p.round{
           proposals[p] = valid_values[p];
         } ->
         Skip
       } else {
         update_proposal_to_new_value_for_process_in_round.p.round{
```

```
            if (honest_processes[p] == true) {
              proposals[p] = DECISION_T;
            } else {
              proposals[p] = DECISION_F;
            }
          } ->
          Skip
        };
        BroadcastProposalMessage(p, rounds[p], proposals[p], valid_rounds[p])
      } else {
        ScheduleOnTimeoutPropose(p, rounds[p])
      }
    };

ScheduleOnTimeoutPropose(p, round) =
  [BOUND_DELTA > INIT_TIMEOUT_PROPOSE + round * TIMEOUT_DELTA || honest_processes[round % N] != honest_processes[p]]
  OnTimeoutPropose(p, round)
  []
  [honest_processes[round % N] == honest_processes[p]]
  Skip;

UponProposalValue(p) =
  [
    (
      p == 0 ||
      (
        rounds[p - 1] == rounds[p] &&
        steps[p - 1] >= PREVOTE
      )
    ) &&
    decisions[p] == NIL &&
    steps[p] == PROPOSE &&
    (
      message_log.ContainsProposal(all_messages, rounds[p], -1)  ||
      message_log.ContainsProposalAndSufficientPrevotesForPrevoting(all_messages, rounds[p])
    )
  ]
  ifa (
    (
```

```
108         (honest_processes[p] == true && message_log.
    GetProposalValue(all_messages, rounds[p]) ==
    DECISION_T) ||
109         (honest_processes[p] == false && message_log.
    GetProposalValue(all_messages, rounds[p]) ==
    DECISION_F)
110       ) &&
111       (
112         (
113           message_log.ContainsProposal(all_messages, rounds
    [p], -1) &&
114           (locked_rounds[p] == -1 || locked_values[p] ==
    message_log.GetProposalValue(all_messages, rounds[p])
    )
115         ) ||
116         (
117           message_log.
    ContainsProposalAndSufficientPrevotesForPrevoting(
    all_messages, rounds[p]) &&
118           (locked_rounds[p] <= message_log.
    GetProposalValidRound(all_messages, rounds[p]) ||
    locked_values[p] == message_log.GetProposalValue(
    all_messages, rounds[p]))
119         )
120       )
121     ) {
122       BroadcastPrevoteMessage(p, rounds[p], message_log.
      GetProposalValue(all_messages, rounds[p]))
123     } else {
124       BroadcastPrevoteMessage(p, rounds[p], NIL)
125     };
126     update_step_for_process_in_round.p.rounds[p]{
127       steps[p] = PREVOTE;
128     } ->
129     Tendermint();
130
131   UponSufficientPrevoteAny(p) =
132     [
133       (
134         p == 0 ||
135         (
136           rounds[p - 1] == rounds[p] &&
137           steps[p - 1] >= PRECOMMIT
138         )
139       ) &&
```

```
    decisions[p] == NIL &&
    (effective_rounds[p] / 4) % 2 == 0 &&
    steps[p] == PREVOTE &&
    message_log.ContainsSufficientPrevotes(all_messages,
   rounds[p], MESSAGE_LOG_DEFAULT_VALUE)
  ]
  enable_first_flag_in_effective_round_for_process.p{
    effective_rounds[p] = effective_rounds[p] + 4;
  } ->
  ScheduleOnTimeoutPrevote(p, rounds[p]);
  Tendermint();

ScheduleOnTimeoutPrevote(p, round) =
  [BOUND_DELTA > INIT_TIMEOUT_PREVOTE + round *
    TIMEOUT_DELTA || honest_processes[round % N] !=
    honest_processes[p]]
  OnTimeoutPrevote(p, round)
  []
  [honest_processes[round % N] == honest_processes[p]]
  Skip;

UponSufficientPrevoteValue(p) =
  [
    (
      p == 0 ||
      (
        rounds[p - 1] == rounds[p] &&
        steps[p - 1] >= PRECOMMIT
      )
    ) &&
    decisions[p] == NIL &&
    (effective_rounds[p] / 2) % 2 == 0 &&
    steps[p] >= PREVOTE &&
    message_log.
   ContainsProposalAndSufficientPrevotesForPrecommitting
   (all_messages, rounds[p]) &&
    (
      (honest_processes[p] == true && message_log.
   GetProposalValue(all_messages, rounds[p]) ==
   DECISION_T) ||
      (honest_processes[p] == false && message_log.
   GetProposalValue(all_messages, rounds[p]) ==
   DECISION_F)
    )
  ]
```

```
176   enable_second_flag_in_effective_round_for_process.p{
177     effective_rounds[p] = effective_rounds[p] + 2;
178   } ->
179   ifa (steps[p] == PREVOTE) {
180
      update_locked_value_and_locked_round_for_process_in_round
      .p.rounds[p]{
181       locked_values[p] = message_log.GetProposalValue(
      all_messages, rounds[p]);
182       locked_rounds[p] = rounds[p];
183     } ->
184     BroadcastPrecommitMessage(p, rounds[p], message_log.
      GetProposalValue(all_messages, rounds[p]));
185     update_step_for_process_in_round.p.rounds[p]{
186       steps[p] = PRECOMMIT;
187     } ->
188     Skip
189   };
190   update_valid_value_and_valid_round_for_process_in_round
      .p.rounds[p]{
191     valid_values[p] = message_log.GetProposalValue(
      all_messages, rounds[p]);
192     valid_rounds[p] = rounds[p];
193   } ->
194   Tendermint();
195
196 UponSufficientPrevoteNil(p) =
197   [
198     (
199       p == 0 ||
200       (
201         rounds[p - 1] == rounds[p] &&
202         steps[p - 1] >= PRECOMMIT
203       )
204     ) &&
205     decisions[p] == NIL &&
206     steps[p] == PREVOTE &&
207     message_log.ContainsSufficientPrevotes(all_messages,
      rounds[p], NIL)
208   ]
209   BroadcastPrecommitMessage(p, rounds[p], NIL);
210   update_step_for_process_in_round.p.rounds[p]{
211     steps[p] = PRECOMMIT;
212   } ->
213   Tendermint();
```

```
214
215 UponSufficientPrecommitAny(p) =
216   [
217     (
218       p == 0 ||
219       rounds[p - 1] > rounds[p]
220     ) &&
221     decisions[p] == NIL &&
222     effective_rounds[p] % 2 == 0 &&
223     message_log.ContainsSufficientPrecommits(all_messages
      , rounds[p], MESSAGE_LOG_DEFAULT_VALUE)
224   ]
225   enable_third_flag_in_effective_round_for_process.p{
226     effective_rounds[p] = effective_rounds[p] + 1;
227   } ->
228   ScheduleOnTimeoutPrecommit(p, rounds[p]);
229
230 ScheduleOnTimeoutPrecommit(p, round) =
231   [(BOUND_DELTA > INIT_TIMEOUT_PRECOMMIT + round *
      TIMEOUT_DELTA || message_log.
      ContainsProposalAndSufficientAdversePrecommits(
      all_messages, round) || honest_processes[round % N] =
      = false) && round < 2]
232   OnTimeoutPrecommit(p, round)
233   []
234   [(!message_log.
      ContainsProposalAndSufficientAdversePrecommits(
      all_messages, round) && honest_processes[round % N] =
      = true) || round == 2]
235   Tendermint();
236
237 UponSufficientPrecommitValue(p) =
238   [
239     (
240       p == 0 ||
241       honest_processes[p - 1] == false ||
242       rounds[p - 1] > rounds[p] ||
243       decisions[p - 1] != NIL
244     ) &&
245     decisions[p] == NIL &&
246     message_log.ContainsProposalAndSufficientPrecommits(
      all_messages) &&
247     (
248       (honest_processes[p] == true && message_log.
      GetCommitReadyValue(all_messages) == DECISION_T) ||
```

```
249         (honest_processes[p] == false && message_log.
      GetCommitReadyValue(all_messages) == DECISION_F)
250       )
251     ]
252
      update_decision_to_commit_ready_value_for_process_in_round
      .p.rounds[p]{
253       decisions[p] = message_log.GetCommitReadyValue(
      all_messages);
254 //      message_log.Clear();
255     } ->
256     Tendermint();
257
258 UponSufficientMessageAny(p) =
259     [
260       decisions[p] == NIL &&
261       message_log.GetLatestRoundWithSufficientMessages() >
      rounds[p]
262     ]
263     StartRound(p, message_log.
      GetLatestRoundWithSufficientMessages());
264     Tendermint();
265
266 OnTimeoutPropose(p, round) =
267     ifa (round == rounds[p] && steps[p] == PROPOSE) {
268       BroadcastPrevoteMessage(p, round, NIL);
269       update_step_for_process_in_round.p.round{
270         steps[p] = PREVOTE;
271       } ->
272       Skip
273     };
274
275 OnTimeoutPrevote(p, round) =
276     ifa (round == rounds[p] && steps[p] == PREVOTE) {
277       BroadcastPrecommitMessage(p, round, NIL);
278       update_step_for_process_in_round.p.round{
279         steps[p] = PRECOMMIT;
280       } ->
281       Skip
282     };
283
284 OnTimeoutPrecommit(p, round) =
285     ifa (decisions[p] == NIL && round == rounds[p]) { //
      decisions[p] == NIL is not in the originally proposed
       algorithm
```

```
    StartRound(p, round + 1)
  };
  Tendermint();

OnTimeoutPrevoteManual() =
  [
    (
      (effective_rounds[0] / 4) % 2 == 1 &&
      steps[0] == PREVOTE &&
      message_log.ContainsAllPrevotesWithoutMajorityValue
    (all_messages, rounds[0])
    ) ||
    (
      (effective_rounds[1] / 4) % 2 == 1 &&
      steps[1] == PREVOTE &&
      message_log.ContainsAllPrevotesWithoutMajorityValue
    (all_messages, rounds[1])
    ) ||
    (
      (effective_rounds[2] / 4) % 2 == 1 &&
      steps[2] == PREVOTE &&
      message_log.ContainsAllPrevotesWithoutMajorityValue
    (all_messages, rounds[2])
    ) ||
    (
      (effective_rounds[3] / 4) % 2 == 1 &&
      steps[3] == PREVOTE &&
      message_log.ContainsAllPrevotesWithoutMajorityValue
    (all_messages, rounds[3])
    )
  ]
  ifa (
    steps[0] == PREVOTE &&
    message_log.ContainsAllPrevotesWithoutMajorityValue(
   all_messages, rounds[0])
  ) {
    OnTimeoutPrevote(0, rounds[0])
  };
  ifa (
    steps[1] == PREVOTE &&
    message_log.ContainsAllPrevotesWithoutMajorityValue(
   all_messages, rounds[1])
  ) {
    OnTimeoutPrevote(1, rounds[1])
  };
```

```
  ifa (
    steps[2] == PREVOTE &&
    message_log.ContainsAllPrevotesWithoutMajorityValue(
   all_messages, rounds[2])
  ) {
    OnTimeoutPrevote(2, rounds[2])
  };
  ifa (
    steps[3] == PREVOTE &&
    message_log.ContainsAllPrevotesWithoutMajorityValue(
   all_messages, rounds[3])
  ) {
    OnTimeoutPrevote(3, rounds[3])
  };
  Tendermint();

OnTimeoutPrecommitManual() =
  [
    (
      effective_rounds[0] % 2 == 1 &&
      steps[0] == PRECOMMIT &&
      message_log.
   ContainsAllPrecommitsWithoutMajorityValue(
   all_messages, rounds[0])// the reason why this cannot
    be moved out of brackets to make it common to all
   disjuncts is because some other processes may have
   already proceeded to the next round up till PREVOTE
   stage
    ) ||
    (
      effective_rounds[1] % 2 == 1 &&
      steps[1] == PRECOMMIT &&
      message_log.
   ContainsAllPrecommitsWithoutMajorityValue(
   all_messages, rounds[1])
    ) ||
    (
      effective_rounds[2] % 2 == 1 &&
      steps[2] == PRECOMMIT &&
      message_log.
   ContainsAllPrecommitsWithoutMajorityValue(
   all_messages, rounds[2])
    ) ||
    (
      effective_rounds[3] % 2 == 1 &&
```

```
      steps[3] == PRECOMMIT &&
      message_log.
    ContainsAllPrecommitsWithoutMajorityValue(
    all_messages, rounds[3])
     )
  ]
  ifa (
    steps[0] == PRECOMMIT &&
    message_log.ContainsAllPrecommitsWithoutMajorityValue
    (all_messages, rounds[0])
  ) {
    OnTimeoutPrecommit(0, rounds[0])
  };
  ifa (
    steps[1] == PRECOMMIT &&
    message_log.ContainsAllPrecommitsWithoutMajorityValue
    (all_messages, rounds[1])
  ) {
    OnTimeoutPrecommit(1, rounds[1])
  };
  ifa (
    steps[2] == PRECOMMIT &&
    message_log.ContainsAllPrecommitsWithoutMajorityValue
    (all_messages, rounds[2])
  ) {
    OnTimeoutPrecommit(2, rounds[2])
  };
  ifa (
    steps[3] == PRECOMMIT &&
    message_log.ContainsAllPrecommitsWithoutMajorityValue
    (all_messages, rounds[3])
  ) {
    OnTimeoutPrecommit(3, rounds[3])
  };
  Tendermint();

Validator(p) =
  UponProposalValue(p) []
  UponSufficientPrevoteAny(p) []
  UponSufficientPrevoteValue(p) []
  UponSufficientPrevoteNil(p) []
  UponSufficientPrecommitAny(p) []
  UponSufficientPrecommitValue(p) []
  UponSufficientMessageAny(p);

```

```
397 TendermintBootstrap() =
398   StartRound(0, 0);
399   StartRound(1, 0);
400   StartRound(2, 0);
401   StartRound(3, 0);
402   Tendermint();
403
404 Tendermint() =
405   Validator(0) []
406   Validator(1) []
407   Validator(2) []
408   Validator(3) []
409   OnTimeoutPrevoteManual() []
410   OnTimeoutPrecommitManual();
411
412 #assert TendermintBootstrap() deadlockfree;
413
414 #define four_processes_are_honest (
415   honest_processes[0] == true &&
416   honest_processes[1] == true &&
417   honest_processes[2] == true &&
418   honest_processes[3] == true
419 );
420 #define three_processes_are_honest (
421   honest_processes[0] == true &&
422   honest_processes[1] == true &&
423   honest_processes[2] == true
424 );
425
426 #define four_processes_make_true_decisions (
427   decisions[0] == DECISION_T &&
428   decisions[1] == DECISION_T &&
429   decisions[2] == DECISION_T &&
430   decisions[3] == DECISION_T
431 );
432 #define three_processes_make_true_decisions (
433   decisions[0] == DECISION_T &&
434   decisions[1] == DECISION_T &&
435   decisions[2] == DECISION_T
436 );
437
438 #assert TendermintBootstrap() |= [] <>
      four_processes_make_true_decisions;
439 #assert TendermintBootstrap() |= [] <>
      three_processes_make_true_decisions;
```

```
#assert TendermintBootstrap() reaches
   four_processes_make_true_decisions;
#assert TendermintBootstrap() reaches
   three_processes_make_true_decisions;
#assert TendermintBootstrap() nonterminating;
```

CSP# Model 1.1. Final Version of CSP# Model (by Us) of Tendermint Consensus Algorithm

2 Optimization Strategy Principles

Prior to delving into the justification of the validity and correctness, of the two pivotal optimizations implemented at the specification level in our model to tackle the intractability of model-checking a faithfully translated distributed consensus protocol such as Tendermint, which is by nature highly concurrent, and consequently has a resulting state space too extensive to be exhaustively explored within a finite duration using the capabilities of contemporary verification tools like PAT, while ensuring that no potential counterexamples that can be identified in the full state space are omitted from the reduced model, we wish to first furnish the readers with the requisite knowledge to fully grasp the discussions that ensue in the forthcoming section § 3.

2.1 Partial Order Reduction.

The connection between the interleaving semantics based upon which an interleaved model of computation, typically used in describing, modelling, and verifying asynchronous concurrent systems, is interpreted, distinguishing between executions that solely differ in the sequence of independent transitions but are often considered equivalent due to identical information conveyed, and the partial order semantics which dictates an order on events sometimes referred to as "causal order"[6] where certain events must precede others, while independent events unconstrained by this "causality relation" may manifest in any order, or rather, appear unordered, is well leveraged by the so-called ***partial order reduction*** technique, to allow a partial order execution to represent multiple interleaving executions, given that the formal specification of a system often fails to differentiate between interleaving executions that are the well-founded *linearizations* of the partial order execution, thereby mitigating the well-known *state space explosion* problem that originates from the exponential growth rate with respect to the number of processes, of the number of states of a concurrent system, by reducing the resulting state space, predominantly derived from the concurrent events being modelled in all conceivable orders relative to one another, to a more compact form (directly built on the fly without the need to construct a full state space first), which merely encompasses representatives for each equivalence class of the executions, but meanwhile guarantees the existence of at least one counterexample for the verified property in the reduced state space if one (or more) is present in the full state space.

As this technique is usually implemented as part of the model checking algorithm which operates over the underlying model representations in the verification engine, we therefore ground our following discussions on a transition system modelled as a Kripke structure $\mathcal{K} = \langle S, I, R, L, AP \rangle$ where a transition $\alpha \in R$ is considered *enabled* in a state $s \in S$ if there exists another state $s' \in S$ such that $(s, s') \in R$, indicating that $\alpha(s, s')$ is valid, and moreover if s' is the only state to which s can transition, or in other words, if the set denoted by $enabled(s)$ which contains all transitions *enabled* from s only has a single element, then α is said to be a *deterministic transition*, in which case α can also be treated as a partial function applicable to states instead of a relation, expressed as $s' = \alpha(s)$ rather than $\alpha(s, s')$. Additionally, we also assume an execution path which is a maximal sequence of states depicting serialized transitions (in some order) occurring concurrently in a system, is always infinite, where a finite path that has reached either a termination or a deadlock (depending on whether it terminates at a desired state or ends prematurely), can always be extended by repeating its last state indefinitely, through a new self-transition which becomes enabled when all others are disabled yet results in no change to the final state. With these assumptions in place, we are now well-equipped to present some of the definitions [2,3] that can assist in reasoning about the correctness of this technique when applied to a system modelled into a state transition system such as a Kripke structure.

To formalize the definition of *reorderable* transitions which need to be identified when selecting a restricted number of orderings for analysis, we first introduce the key concept of *independence relation* between transitions as follows.

Definition 1 (Independence Relation)**.** The *independence relation* $IR \subseteq R \times R$ is a symmetric and anti-reflexive relation on transitions where each pair of *independent* transitions $(\alpha, \beta) \in IR$ satisfies the following two conditions for each state $s \in S$:

- *Enabledness*: If $\alpha, \beta \in enabled(s)$, then $\alpha \in enabled(\beta(s))$ and $\beta \in enabled(\alpha(s))$. [Two independent transitions enabled in a given state cannot disable each other.]
- *Commutativity*: If $\alpha, \beta \in enabled(s)$, then $\alpha(\beta(s)) = \beta(\alpha(s))$. [The execution of two enabled independent transitions in any order (whose feasibility is guaranteed by the *enabledness* condition) results in the same global state.]

We can then deductively infer that the *dependency relation* $DR = (R \times R) \setminus IR$ that encompasses all *dependent* transitions (*i.e.*, transitions that are not *independent*) is therefore the complement of the *independence relation*, excluding the cases such as, for instance, two local transitions in different processes that do not mutate the global state, and transitions corresponding to asynchronous send and receive operations (to and from different processes), which are both deemed independent.

However, considering that certain checked properties may distinguish between intermediate states of execution sequences that only differ in the order of their constituent transitions that are independent of each other, in the sense that the verification outcome may vary depending on these seemingly equivalent reordered

execution sequences, we thus propose the second key notion of *invisible* transitions, which serve to elucidate the criteria by which a specification discriminates between two states, as delineated below.

Definition 2 (Invisible Transitions)**.** A transition $\alpha \in R$ is considered *invisible* with respect to some subset AP' of the entire set of atomic propositions AP (*i.e.*, $AP' \subseteq AP$) if its execution between any two states does not culminate in a change in their labellings (*i.e.*, for each pair of states $s, s' \in S$ such that $s' = \alpha(s)$, $L(s) \bigcap AP' = L(s') \bigcap AP'$).

Now, we are adequately prepared to rigorously formalize the notion of *equivalence* among sequences of executions with respect to the properties to be verified in LTL formulas, by introducing the third key concept defined as follows.

Definition 3 (Stuttering Equivalence of Paths)**.** Two infinite paths are *equivalent up to stuttering* if they share the identical state labellings after the *stutter removal operator* is applied to collapse each of their maximal sequences of identically labelled states into a single occurrence of a state with that labelling. To put it more formally, two infinite paths $\sigma = s_0, s_1.s_2, ...$ and $\rho = r_0, r_1, r_2, ...$, are *stuttering equivalent*, denoted by $\sigma \equiv_{st} \rho$, if two infinite sequences of integers $0 = i_0 < i_1 < ...$ and $0 = j_0 < j_1 < ...$ can be defined such that $\forall k \geq 0, L(s_{i_k}) = L(s_{i_k+1}) = L(s_{i_{k+1}-1}) = L(r_{j_k}) = L(r_{j_k+1}) = L(r_{j_{k+1}-1})$, where the indices i_k and j_k refer to the starting point of identically labelled subsequence of states in each of these two paths respectively.

Definition 4 (Stuttering Invariance of LTL Formulas)**.** An LTL$_{-X}$ formula ψ (without the use of any "next" operator) is *invariant under stuttering* if for any two *stuttering equivalent* paths σ and σ' (*i.e.*, $\sigma \equiv_{st} \sigma'$), $\sigma \models \psi$ iff $\sigma' \models \psi$ holds.

Definition 5 (Stuttering Equivalence of State Transition Systems)**.** By a simple extension of the notion defined in Definition 3 over paths, two state transition systems M and M' are considered *stuttering equivalent* iff

- there exists a path σ' starting from an initial state of M' for each path σ starting from an initial state of M such that $\sigma \equiv_{st} \sigma$‘ holds, and
- there exists a path σ starting from an initial state of M for each path σ' starting from an initial state of M' such that $\sigma' \equiv_{st} \sigma$ holds.

Based on the definition of *stuttering equivalence* of state transition systems presented in Definition 5 and the definition of *stuttering invariance* of LTL$_{-X}$ formulas shown in Definition 4, we are able to draw the conclusion that if two state transition systems M and M' are *stuttering equivalent*, then $M \models \psi$ iff $M' \models \psi$ holds for any given LTL$_{-X}$ formula ψ, which should serve as the justification of the validity of the use of partial order reduction technique to generate a structure *stuttering equivalent* to the original state transition system, in enforcing no missing counterexamples from the reduced state space if they are indeed present in the full state space.

In light of the objectives of this section, we will not delve into the algorithms underpinning partial order reduction, which are used to compute a subset $ample(s)$

of the enabled transitions $enabled(s)$ from each state s to be expanded on, rather than all the transitions in $enabled(s)$[1], but instead simply characterize the subset $ample(s)$ through a series of conditions, the satisfaction of which guarantees the existence of at least one representative in the generated state space, stuttering equivalent to each execution sequence in the full state space, and hence the preservation of at least one counterexample in the reduced state space if one is present in the full state space, where a formal proof of this argument can be found in [2,3] if interested.

There are in total four conditions listed below, that need to be satisfied for a reduction to be considered correct and valid.

- **C0**: $ample(s) = \emptyset$ iff $enabled(s) = \emptyset$. [*Emptiness condition* guarantees the eventual progress to be made by the algorithm with reduction if the standard search would.]
- **C1**: On any execution sequence starting from some state s of the full state graph, a transition dependent on a given transition from $ample(s)$ cannot be executed before that transition from $ample(s)$ gets executed. [*Ample Decomposition condition* guarantees that any path from the full state space but not part of the reduced state space can be converted, based on the properties of independent transitions, into a path included in the reduced state space, thereby preventing the exclusion of crucial paths needed for verification.]
- **C2**: If a state s is not *fully expanded*, then every transition $\alpha \in ample(s)$ must be *invisible*. [*Invisibility condition* guarantees the *stuttering equivalence* between the transformed path from the reduced state space and the original one from the full state space, thereby keeping the truth value of the specification from being affected by this reduction.]
- **C3**: If a state s is not *fully expanded*, then no transition $\alpha \in ample(s)$ will appear on the search stack of the DFS search algorithm, or alternatively, every cycle in the reduced state space must contain at least one *fully expanded* state. [*Cycle Closing condition* guarantees absence of indefinitely deferred transitions in a reduced state space.]

To demonstrate the reduction achieved by solely leveraging conditions **C1** and **C2**, consider two concurrent processes: one with two sequentially executed local/internal transitions α_1 and α_2, and the other similarly with two sequentially executed local/internal transitions β_1 and β_2, whose joint full state space is illustrated on the left-hand side of Figure 1, application of partial order reduction to which results in an ample set from the initial state comprised of merely a single transition α_1 followed by a state with only α_2 enabled, and then afterward a state consisting of only β_1 followed by another state with the remaining β_2 enabled, as shown on the right-hand side of Figure 1.

[1] A *full expansion* of $enabled(s)$ is required only if the property being verified distinguishes among the intermediate states of *stuttering equivalent* execution sequences, or if removing one of these states causes certain successor states to remain unexplored, ultimately leading to a false negative result.

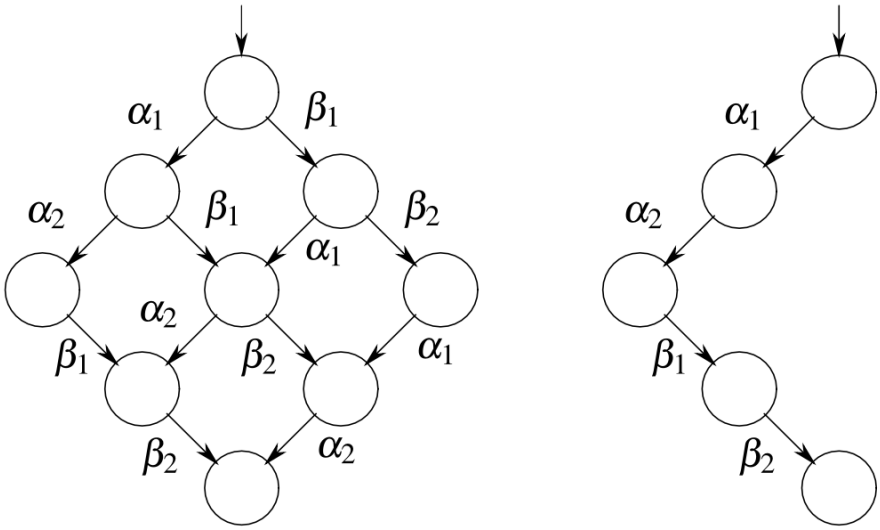

Fig. 1: Partial Order Reduction Example with Ample Sets [3]

2.2 Symmetry Reduction.

Inherent structural symmetries, such as replicated components present in many concurrent systems, where behaviours exhibited by multiple processes are indistinguishable or interchangeable under identical conditions with respect to the properties under verification, are often overlooked and not taken advantage of, despite their potential to significantly mitigate the *state space explosion* problem by allowing state permutations that differ only in the arrangement of symmetric entities to be treated as equivalent, thereby enabling the ***symmetry reduction***, as an orthogonal technique that targets a different axis of state space redundancy in contrast to partial order reduction which focuses on the minimization of the number of interleavings of independent transitions, to collapse such symmetric states, or equivalence classes of states termed *orbits*, into one single representative from each orbit for exploration during the model checking process, without compromising the soundness and completeness of the analysis by ensuring that if a counterexample to the specified property exists in the full state space, a corresponding trace will also manifest within the reduced space, while eliminating unnecessary computations to effectively manage the exponential growth of the state space during the verification of complex asynchronous concurrent systems.

Given the same assumptions on the transition system model, specifically the Kripke structure $\mathcal{K} = \langle S, I, R, L, AP \rangle$ whose semantics have previously been defined and analysed in the context of partial order reduction, along with its associated assumptions regarding enabled transitions, deterministic behaviours, and the infinite nature of execution paths, where finite paths are treated extensible by looping the last state indefinitely, which serves as a common framework for

both reduction techniques, no further reiteration of them will be provided here to maintain succinctness and enforce consistency across our discussions on state space reduction, allowing us to directly focus on the definitions [1,4,5] presented as follows that can aid in reasoning about the correctness of symmetry reduction techniques applied at the modelling language level, to a distributed protocol modelled into a state transition system such as a Kripke structure in this part of our work.

To formalize the definitions of *symmetric* states/paths, which need to be identified when selecting only one representative from each equivalence class of them for analysis, we first introduce the key concept of *permutation* and then deductively *symmetry group* and *orbit* in a transition system.

Definition 6 (Permutation)**.** A permutation $\sigma : S \to S$ on a set S is a *bijection* of S into itself, or formally $(\forall s \in S : \exists_{=1}\sigma(s) \in S) \wedge (\forall \sigma(s) \in S : \exists_{=1} s \in S)$.

Definition 7 (Symmetry Group)**.** Given a transition system modelled as a Kripke structure $\mathcal{K} = \langle S, I, R, L, AP \rangle$, a subgroup G of the group of all permutations of S denoted $Perm(S)$ is called a *symmetry group* of $\mathcal{K}$ if for all permutations σ in G, they preserve the transition relation R, or formally $(s_1, s_2) \in R$ iff $(\sigma(s_1), \sigma(s_2)) \in R$, and every permutation $\sigma \in G$ is called an *automorphism* of $\mathcal{K}$.

Definition 8 (Equivalence Relation & Orbit)**.** Given a transition system modelled as a Kripke structure $\mathcal{K} = \langle S, I, R, L, AP \rangle$, a symmetry group G acting on $\mathcal{K}$ defines an *equivalence relation* $\equiv$ on S such that $s_1 \in S \equiv s_2 \in S$ if $\exists \sigma \in G : s_2 = \sigma(s_1)$, and the *equivalence class* $[s] = \{t \mid \exists \sigma \in G : \sigma(s) = t\}$ of s under $\equiv$ is called the *orbit* of s under G, whose representative is denoted as $rep([s]) \in [s]$.

We then can define the *quotient model* of a transition system, which will be useful in proving the correctness of our adapted reduction technique later in this section, as follows.

Definition 9 (Quotient Structure)**.** Given a transition system modelled as a Kripke structure $\mathcal{K} = \langle S, I, R, L, AP \rangle$ and a symmetry group G acting on $\mathcal{K}$, a quotient structure that models the transition system for $\mathcal{K}$ modulo G is a Kripke structure $\mathcal{K}_G = \langle S_G, I_G, R_G, L_G, AP \rangle$ where

- $S_G = \{[s] \mid s \in S\}$,
- $I_G = \{[s_0] \mid s_0 \in I\}$,
- $R_G = \{([s], [t]) \mid (s, t) \in R\}$,
- $L_G : S_G \to 2^{AP}$ such that $\forall s \in S : L_G([s]) = L(rep([s])) = L(s)$

Another crucial condition that needs to be satisfied for the symmetry reduction to be considered valid and correct concerns the *invariance* of the atomic propositions under verification.

Definition 10 (Invariance Group)**.** Given a transition system modelled as a Kripke structure $\mathcal{K} = \langle S, I, R, L, AP \rangle$, a symmetry group G acting on $\mathcal{K}$ is an *invariance group* for an atomic proposition p if and only if the set of states labelled

by p is closed under the application of all the permutations of G, or formally if and only if the following condition holds: $\forall \sigma \in G : \forall s \in S : p \in L(s) \Leftrightarrow p \in L(\sigma(s))$, implying that $\forall \sigma \in G : \forall s \in S : L(s) = L(\sigma(s))$.

Lastly, there is one more auxiliary definition that needs to be presented as follows for the theorem at the end of this section to be properly articulated.

Definition 11 (Bisimulation Relation)**.** Given two transition systems modelled as Kripke structures $\mathcal{K}_1 = \langle S_1, I_1, R_1, L_1, AP \rangle$ and $\mathcal{K}_2 = \langle S_2, I_2, R_2, L_2, AP \rangle$, a binary relation $\mathcal{B} \subseteq S_1 \times S_2$ is a *bisimulation relation* if $(s_1, s_2) \in \mathcal{B}$ implies that:
- $L_1(s_1) = L_2(s_2)$
- if $(s_1, s_1') \in R_1$, then $\exists s_2' \in S_2 : (s_2, s_2') \in R_2 \wedge (s_1', s_2') \in \mathcal{B}$
- if $(s_2, s_2') \in R_2$, then $\exists s_1' \in S_1 : (s_1, s_1') \in R_1 \wedge (s_1', s_2') \in \mathcal{B}$

In essence, two states s_1 and s_2 are *bisimilar*, denoted $s_1 \sim s_2$, if they are related by some *bisimulation relation*, and deductively two transition systems are *bisimilar* or *bisimulation equivalent*, denoted $\mathcal{K}_1 \sim \mathcal{K}_2$, if their state spaces behave identically, or more specifically, starting from corresponding states in each state space, they can always simulate each other's transitions, exhibiting no observable differences in their sequences of actions as far as the properties under verification are concerned.

Theorem 1. *Given a transition system modelled as a Kripke structure* $\mathcal{K} = \langle S, I, R, L, AP \rangle$ *and its quotient structure* $\mathcal{K}_G = \langle S_G, I_G, R_G, L_G, AP \rangle$ *with respect to an invariance group* G *for all the atomic propositions* $p \in AP$ *occurring in every possible symmetric LTL formula* ϕ*,* $\mathcal{K}_G$ *is* bisimulation equivalent *to* $\mathcal{K}$*, and hence* $\forall s \in S : \mathcal{K}, s \models \phi \Leftrightarrow \forall [s]_G \in S_G : \mathcal{K}_G, [s]_G \models \phi$*.*

Based on the theorem presented in Theorem 1 whose formal proof can be found in [1,4] if interested, we are able to draw the conclusion that if two state transition systems $\mathcal{K}$ and $\mathcal{K}_G$ are *bisimulation equivalent*, then $\mathcal{K} \models \phi$ iff $\mathcal{K}_G \models \phi$ holds for any given LTL formula ϕ, which should serve as the justification of the validity of the use of symmetry reduction technique to generate a quotient state transition system *bisimulation equivalent* to the original full state transition system, in enforcing no missing counterexamples from the reduced state space if they are indeed present in the full state space.

In view of the objectives of this section, we will not explore further the algorithms underpinning symmetry reduction, especially the *canonicalization function* ζ used for on-the-fly mapping from each state s into its unique representative $\zeta(s) = rep([s])$ of the equivalence class $[s]$, which mainly facilitates the computation of the orbit relation (*i.e.*, determination of whether two states belong to the same orbit or not), as a direct method often leveraged in building the quotient model, but proven to be as hard as the graph isomorphism problem by Clarke [1], which is NP-complete.

To demonstrate the reduction achieved by the exploitation of symmetries, consider two concurrent processes with symmetric roles, each comprised of only a single actionable operation independent of each other after initialization, whose joint full state space is illustrated in Figure 2, application of symmetry

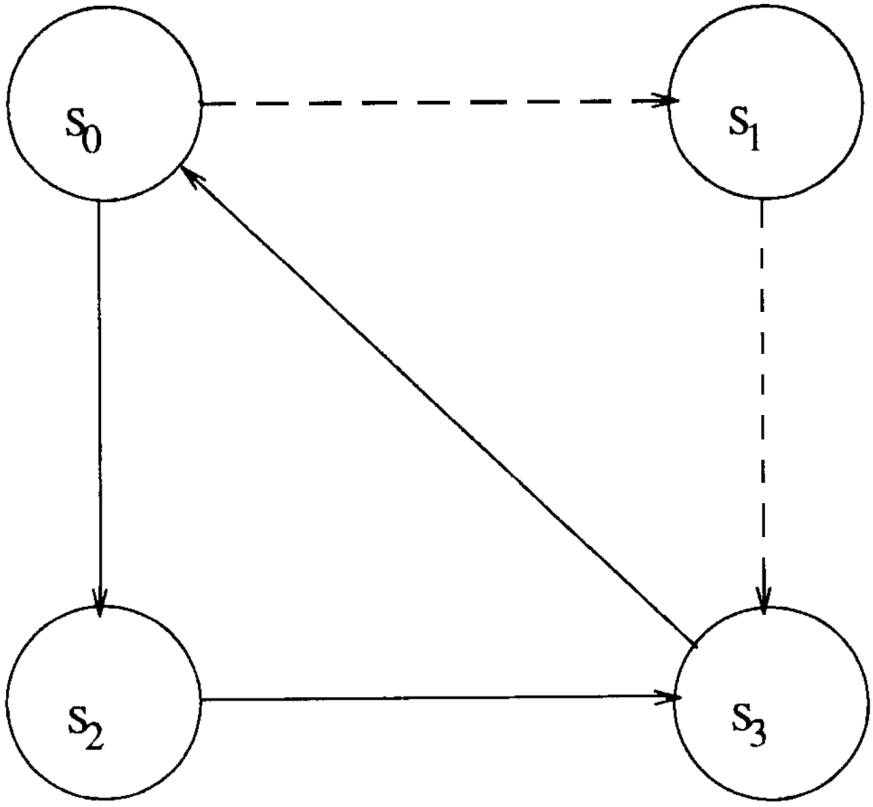

Fig. 2: Symmetry Reduction Example with Orbit Relations [4]

reduction to which results in a symmetry group G consisting of only a single permutation $\sigma = \{(s_0, s_0), (s_1, s_2), (s_2, s_1), (s_3, s_3)\}$, and hence an orbit relation of $[s_2] = \{s_1, s_2\}$ under G with $rep([s_1]) = rep([s_2]) = s_2$, to be computed and leveraged to omit the state s_1 as well as all transitions going through s_1 from the final state space for exploration, culminating in a quotient model of the original state transition system.

3 Optimization Strategy Review

We are now well-prepared to examine the strategies implemented in our model to minimize the state space for exploration, and most importantly, to justify their validity and correctness based on the theorems and principles outlined in § 2. It is imperative to acknowledge that the primary optimization strategies that significantly contributes to the transformation of the model from an inoperative state to an operative state, or to put it more straightforward, those techniques that

effectively achieve an exponential reduction in the size of the state space of the model to manageable dimensions, for feasible exploration to be concluded within acceptable time limits without running out of memory installed on an average PC, are **partial order reduction** and **role-based symmetry reduction** which used to be implemented as part of the model-checking algorithm in the verification engine for automation of this process prone to errors if performed manually by engineers who are unfamiliar with this field of study, but are instead applied to the specification level by us this time to overcome the limitations of these techniques in detecting the partial-order/symmetry structures inherent in the system design under certain assumptions that are not readily convertible into corresponding constructs in a model for the verification engine to recognize automatically, thereby causing the modern verification tool to fail in activating these techniques correctly when choosing the appropriate the optimization techniques to undertake, among others enumerated below that are considered trivial and therefore less important concerning the extent of state space reduction they managed to achieve.

3.1 Partial Order Reduction.

The implementation of this technique manifests itself in the way in which two processes are defined, namely the process $TendermintBootstrap$ at Line 397 and the process $Tendermint$ at Line 404, which are to be separately discussed and reasoned about in this section, with insights into the rationale behind our application of this method in each particular scenario provided first, followed by a detailed examination of the characteristics of the reduced state space with respect to the aforementioned four conditions (*i.e.*, in terms of why the resulting ample sets meet these conditions) demonstrated next.

A separation into two distinct processes, namely $TendermintBootstrap$ and $Tendermint$, as our model does, is favored over the more intuitive approach of combing them into a single process of interleaved execution among all participating validators, each represented by its own process $Validator$ beginning with a $StartRound$ function and followed by a general choice of those processes denoting upon rules (*i.e.*, implying potential modifications required to in fact all of these three processes including $Validator$, $TendermintBootstrap$, and $Tendermint$), given that this modelling approach greatly minimized, by an order of magnitude, the state space by reducing the complexity introduced by the interleaving operator, which attempts to generate all conceivable orders of executions, thereby successfully addressing the issues of timeout and out-of-memory errors encountered previously with an earlier version of the model built based on this naive way of modelling *concurrency* when running the verification tool over the underlying representation of the old model. The intuition behind the decision of imposing a pre-defined order into the constituent processes of $TendermintBootstrap$ and replacing interleaving operators with general choice operators in composing the constituent processes of $Tendermint$ stems from the observation that nearly all transitions (*i.e.*, operations at each step) are *local/internal* (excluding those pure events that do not carry accompanying data operations, and therefore do not introduce side effects to the global state as a result), the execution of which only

mutates local variables without being empowered to disturb the execution of other processes, thereby can never disable each other (*i.e.*, the *enabledness* condition is satisfied), and can be commutated with each other while still leading to the same resulting state at the end (*i.e.*, the *commutativity* condition is satisfied), indicating that these *local* transitions must always be *independent* of each other. Although this argument should be sufficiently convincing, we intend to further dissect these two scenarios into smaller cases for an in-depth analysis of the *stuttering equivalence* between the model built based on the naive approach and the one built with this optimization incorporated, to avoid a lack of academic rigor being argued against this piece of work.

We first take a look at the simpler scenario of partial order reduction applied at the process level in our model specification, which delineates a strict execution sequence imposed on the process *StartRound* (named after the corresponding function) in each validator node. Assuming the identified vulnerability presented in our paper is rectified by adding a conditional check of the round to be advanced into against the current round before its execution, to ensure that a validator node would never revert back to a prior round due to any initial delay in the invocation of $StartRound(0)$ in the affected node, caused by either malicious interference holding it back, or simply a slow, low-spec processor installed in the node, in addition to enforcing the completion of each *StartRouund* function before triggering other upon rules in each validator node, we opt to, instead of adopting the naive approach of using interleaved operators to compose them (*i.e.*, $StartRound(0,0) \;|||\; StartRound(1,0) \;|||\; StartRound(2,0) \;|||\; StartRound(3,0)$), which substantially enlarges the resulting state space, employ sequential composition operators to mandate a fixed execution order (*i.e.*, $StartRound(0,0); StartRound(1,0); StartRound(2,0); StartRound(3,0)$), starting with p_1 (*i.e.*, process 1), followed by p_2 (*i.e.*, process 2), p_3 (*i.e.*, process 3) and then p_4 (*i.e.*, process 4), exactly adhering to the topmost path marked in red in Figure 3[2] if the entire state space is to be visualized into a state machine with each transition indexed by the performing entity of this operation, rather than being decomposed into granular components where each transition is annotated with a single data operation event instead. Note that even though this depiction is said to be a visualization of the full state space with respect to the executions of these four functions running in different nodes, this claim is in fact deceptive in the sense that it intentionally excludes all possible interleaved executions of these functions for clarity, since in reality, when considering the naive approach of modelling concurrent executions among these four processes, a more precise method would entail taking into account all conceivable sequences of events from different processes being interleaved. However, given the reasoning concerning this aspect (*i.e.*, the rationale for omitting parts of the state space representing the interleaved execution of events from the process *StartRound* of different nodes) is elucidated in subsequent paragraphs, we refrain from delving into this

[2]Note that some identical states with the same labellings are intentionally left unaggregated for clarity.

matter here, focusing instead on substantiating why it is adequately sufficient to verify a singular trace among all those depicted in Figure 3.

A careful examination of the definition of the process *StartRound* which is to be executed atomically as a whole (in comparison with its function definition presented in the protocol specification) should reveal that indeed all the operations undertaken are local/internal in the sense that each process only modifies its local variables, except for the leader's broadcast operation that appends the $PROPOSAL$ message to message logs of other nodes, which nevertheless, does not disable the execution of the process *StartRound* of other nodes, and thus also considered independent of each other in this case, implying that regardless of the order in which these four processes from different nodes are executed, all of these paths end up with the same resulting state from which further transitions may be taken to perform operations encompassed in those upon rules. Given this visualization of the state space corresponding to the process $TendermintBootstrap$, we demonstrate the correctness and validity of reducing it to only the topmost path for verification, with respect to the four conditions illustrated in § 2.1 by walking through how each condition is satisfied by each state along the red-highlighted path as follows:

- The condition **C0** is trivially satisfied given that for every state $s \in \{s_0, s_1, s_5, s_{11}\}$ from the reduced state space, $enabled(s) \neq \emptyset \wedge \ ample(s) \neq \emptyset$ holds.
- The condition **C1** is also trivially satisfied since there does not exist any transitions that are dependent on one another, given that operations such as updates to local variables or message broadcasting (by a proposer node) or scheduling of timeout handlers never disable each other (*i.e.*, an operation performed in one node will never render any operation no longer active in another node, implying that the *enabledness* property holds) and arbitrary rearrangement of their relative orders lead to the same resulting state (*i.e.*, regardless of whether the message broadcasting operation is carried out very first or not by a proposer node, another ordinary node is still empowered to non-deterministically decide on if the timeout will be triggered later based on the guard condition specified in the definition of the guarded process $ScheduleOnTimeoutPropose$ to account for the potential message delay due to unstable network, indicating that the *commutativity* property similarly holds), thereby denoting the *independence* of transitions within *StartRound* of different nodes, which consequently suggests the *independence* of the process *StartRound* itself of each node if the set of transitions encapsulated inside each process is to be viewed as a single cohesive unit in its entirety.
- The condition **C2** is satisfied, given that data operations to be undertaken within the process *StartRound* of each node, irrespective of its role being either a proposer node or simply an ordinary validator node, does not involve modifications towards the variable *decisions*, whose value is checked against to verify if the correctness criteria (*i.e.*, both *safety* and *liveness* properties) of this protocol are met or not, thereby denoting the *invisibility* of transitions within *StartRound* of different nodes, which consequently suggests the *invisibility* of the process *StartRound* itself of each node, whose execution

does not result in a change in the labellings of adjacent states with respect to the subset of atomic propositions AP' over the variable *decisions*.
- The condition **C3** is also satisfied, given the absence of any cycles in the reduced state space under the assumption that the identified vulnerability which allows a delayed execution of the process $StartRound(0)$ by each node, likely resulting in a loop back to an earlier round, in which case may form a cycle in the reduced state space and as a result render this condition no longer valid, as none of the ample sets of these four states is fully expanded, is rectified, thereby suggesting the importance of the fix to that identified vulnerability towards the correctness and validity of partial order reduction applied in this place.

This technique has been similarly applied to the event level in our model specification, by mandating the mutually exclusive execution of each upon rule across all nodes involved, thereby simulating a scenario wherein at any given moment, only an upon rule of a single node is allowed to execute, while other nodes are blocked from making progress, similar to the case where a node is granted with a distributed mutex (though implicit in this setting here) so that only the node possessing this mutex is authorized to execute its upon rule, one at a time in an atomic manner as per the protocol specification, while all other nodes must wait for the release of this singular system-wide mutex before they can acquire it to initiate executions of their own upon rules. Under the assumption akin to the one made above for process-level partial order reduction, of a model corresponding to the protocol specification with an amendment rectifying the identified vulnerability, we also choose to, instead of adopting the naive approach of using interleaved operators (*i.e.*, $Validator(0)|||Validator(1)|||Validator(2)|||Validator(3)$) to compose processes, each of which denotes the behaviors (in the form of upon rules) that can be engaged by a validator node, which also notably expands the resulting state space, employ general choice operators (*i.e.*, $Validator(0)[]Validator(1)[]Validator(2)[]$ $Validator(3)$) to enforce mutually exclusive, atomic execution of each upon rule across all nodes, whose implication includes omission from the reduced state space, of all conceivable interleaved executions of upon rules from different nodes. The intuition behind this optimization at the event level is that once two or more upon rules start their executions concurrently, which implies that their guarded expressions, each representing the condition specified in the header of its corresponding upon rule must be evaluated to true, all the operations undertaken are local/internal, incapable of interfering with executions of other nodes, since an upon rule, once activated (*i.e.*, once its guard condition is evaluated to true), cannot be deactivated unless it has been executed at least once, subsequent to which certain rules annotated with "for the first time" become inactive afterward, suggesting that an interleaved execution of upon rules from different nodes can effectively be serialized into an atomic execution of one of them from a node, followed by another one from a different node, and so on.

To better illustrate this concept, a state machine similar to the one presented above for process-level reduction, is constructed and depicted in Figure 4[3] where each transition is annotated with a generalized update operation abstracted from a data operation event this time instead, towards a single variable *var*, indexed by the performing entity (*e.g.*, var_{p_1} represents an update operation performed by the process p_1 towards the variable *var*), given that pure events do not introduce side effects to the internal state of each node, and therefore are ruled out here for brevity. Note that we do not incorporate more than one variable into the state machine due to the complexity involved in embedding more while still keeping the resulting diagram readable, provided that it has now already comprised 16 states with only a single variable involved, although the flexibility of adapting this diagram to accommodate changes to process definitions where updates to multiple variables are required, is respected by the use of dotted lines to represent transitions between states, during which different kinds of events, such as updates to other variables expressed in all feasible permutations, may be integrated to extend the path in between.

In spite of the fact that this state machine is constructed according to the same observation upon which the process-level reduction illustrated above is based, where any data operation event from a node either alter its own local variables, or broadcast messages to other nodes by appending messages to their message logs, which can also be viewed as a form of mutation of variables, though not locally, the reasoning about the correctness of reduction applied at event level is indeed considerably more complex than the one presented previously with respect to those four conditions, since one of these conditions is not fulfilled, yet the essential property that must be upheld by this condition remains intact, as demonstrated in the subsequent proof sketch that follows:

- The condition **C0** is trivially satisfied given that for every state s that must have lied in one of the paths starting from s_0 to s_{15}, and therefore should potentially be part of the reduced state space, $enabled(s) \neq \emptyset \wedge\ ample(s) \neq \emptyset$ holds.
- The condition **C1** is also trivially satisfied since there does not exist any transitions that are dependent on one another, given that operations such as updates to local variables or message broadcasting or scheduling of timeout handlers never disable each other (*i.e.*, an operation performed in one node will never render any operation no longer active in another node, implying that the *enabledness* property holds) and arbitrary rearrangement of their relative orders, including those that are not visualized here but instead subsumed by the dotted lines, lead to the same resulting state (*i.e.*, regardless of whether the message broadcasting operation is carried out before or after another operation by a validator node, another node, whether being the intended recipient or not, is still empowered to non-deterministically decide on if the corresponding timeout, for either $PROPOSAL$, $PREVOTE$, or $PRECOMMIT$ message, will be triggered later based on the guard condition specified in the definition

[3]Note that some identical states with the same labellings are intentionally left unaggregated for clarity.

of the corresponding guarded process, namely $ScheduleOnTimeoutPropose$, $ScheduleOnTimeoutPrevote$ or $ScheduleOnTimeoutPrecommit$, to account for the potential message delay due to unstable network, indicating that the *commutativity* property similarly holds), thereby denoting the *independence* of transitions that represent data operations from upon rules of different nodes.

- The condition **C2** is in fact NOT always satisfied, given that a state from the reduced state space may enable a transition representing an update operation to the variable *decisions* with respect to which our assertion of correctness criteria is defined (*i.e.*, the data operation event contained in the process $UponSufficientPrecommitValue$ at Line 237), likely resulting in a change in the labellings between the two adjacent states connected by this transition, and should therefore be treated as a *visible* transition, whose execution may render the transformed path from the reduced state space and the original one from the full state space no longer *stuttering equivalent*. However, considering the nature of this operation where an update operation in this case can only switch the verification result (*i.e.*, evaluations of the LTL formula [] <> $four\_processes\_make\_true\_decisions$ specified at Line 438 and the LTL formula [] <> $three\_processes\_make\_true\_decisions$ specified at Line 439) from $false$ to $true$ instead of the other way around, and the purpose of verification in detecting a potential counterexample trace that may disprove the correctness of this protocol in terms of *safety* and *liveness*, even if the truth value of the specification may be affected by this reduction as a result of the breach of this condition, the impact remains positive in the sense that no counterexamples from the full state space may be missing from the reduced state space, because a counterexample demonstrating either the non-termination or the lack of safety of the algorithm, will persist in the reduced state space, where a process unable to terminate successfully with a decision made will similarly remain non-terminable in the reduced state space subsequent to the update of the variable *decisions* performed by another process, and a process that concludes with a decision divergent from that of the one who updated its corresponding decision in the variable *decisions*, will nonetheless arrive at an identical decision in the reduced state space, irrespective of whether this *visible* transition is carried out or not, suggesting that the implications brought about by the fulfillment of the condition **C2** also remain relevant even after implementing this event-level reduction, and therefore the condition **C2** can also be deemed satisfied in this case.
- The condition **C3** is also satisfied, given the absence of any cycles in the reduced state space under the assumption that the identified vulnerability which allows a delayed execution of the process *StartRound*(0) by each node, likely resulting in a loop back to an earlier round, in which case may form a cycle in the reduced state space and as a result render this condition no longer valid, as none of the ample sets of these 16 states is fully expanded, is rectified, thereby suggesting the importance of the fix to that identified

vulnerability towards the correctness and validity of partial order reduction applied in this place.

Therefore, we conclude here that the state transition system, when subjected to both process-level and event-level partial order reductions illustrated in this section, is *stuttering equivalent* to the original transition system constructed using naive approaches, indicating that the result of verifying properties against the reduced state transition system must be applicable to the original state transition system as well.

3.2 Symmetry Reduction.

The implementation of this technique manifests itself in the way in which the guarded expressions of processes, each corresponding to an upon rule that reacts to reception of a certain type of messages, are defined, in particular with respect to two dimensions of symmetries, namely *step* and *round*, which are therefore to be separately analysed and reasoned about in this section, similarly with insights into the rationale behind our application of this method to this particular concurrent model first, followed by a detailed examination of the validity and completeness of the reduced state space in proving the correctness of this method.

To illustrate the gist of symmetry reduction applied to the model built specifically for Tendermint with in total four processes involved, a diagram whose states and transitions are labelled in a generalized manner, is presented in Figure 5, which is to be also referenced later when discussing each specific axis of symmetry exploited with those labels concretized. Given a sequence of function executions (*i.e.*, upon rule executions), $func\alpha, func\beta, func\gamma, func\theta$, each being carried out by one of the four processes, regardless of the order in which these four function executions are arranged, any two executions are symmetric with respect to one another, in the sense that their resulting states (as well as all intermediate states) are identical except that the indexes of certain local states are swapped between symmetric processes (*e.g.*, consider the first two execution sequences in the diagram presented in Figure 5, s_4^0 is exactly the same as s_4^1 except that the local state of p_3 is swapped with the local state of p_4, since the roles of p_3 and p_4 are symmetric with respect to one another), implying that if a counterexample trace exists along any execution path that begins with a prefix of one of these execution sequences, the same counterexample must necessarily exist in all other execution sequences, given rise to by other possible permutations of process identifiers, as long as the role invariance is respected, or to put it simply, the properties under verification do not depend on the specific identities of the role instances (*e.g.*, the identities of the exact validators that perform certain operations such as prevote/precommit is not of concern, as long as a majority of them does). Note that every process participating in this consensus procedure assumes the same role of an ordinary validator, regardless of whether it also acts as a proposer in a given round or not, where the distinction between them only lies in the specific operations executed within the function *StartRound*, which have already been separately addressed though through the partial order reduction techniques discussed earlier, and therefore do not need further consideration here.

In addition to the original conditions stipulated for each upon rule in the protocol specification, two extra constraints with respect to two orthogonal dimensions in data values, namely *round* and *step* introduced by us serve as the key to a significant reduction of the state space by an order of magnitude, via imposing a strict total ordering on the executions of upon rules by different processes at different steps within the same round, as well as those executions of upon rules similarly by different processes at potentially same or different steps across different rounds, so that unnecessary exploration of symmetric executions may be avoided to enable a faster termination of the model checking process. A closer inspection of the code definition of each class of upon rules in the model reveals that a given process p is only allowed to

- execute the rule (*i.e.*, $UponProposalValue$) that reacts to the reception of a proposal if it has the smallest index (*i.e.*, $p == 0$) or its preceding process with an index $p-1$ has already taken actions to respond to this event within the same round and hence has advanced the step from $PROPOSE$ to $PREVOTE$ or even $PRECOMMIT$(*i.e.*, $rounds[p-1] == rounds[p]$ && $steps[p-1] >= PREVOTE$), or has proceeded to the next round already (*i.e.*, $rounds[p-1] > rounds[p]$); or
- execute the rules (*i.e.*, $UponSufficientPrevoteAny$ or $UponSufficientPrevoteValue$ or $UponSufficientPrevoteNil$) that react to the reception of sufficient enough prevotes of a certain type if it has the smallest index (*i.e.*, $p == 0$) or its preceding process with an index $p-1$ has already taken actions to respond to this type of event within the same round and hence has advanced the step from $PREVOTE$ to $PRECOMMIT$ (*i.e.*, $rounds[p-1] == rounds[p]$ && $steps[p-1] >= PRECOMMIT$), or has proceeded to the next round already (*i.e.*, $rounds[p-1] > rounds[p]$); or
- execute the rules (*i.e.*, $UponSufficientPrecommitAny$ or $UponSufficientPrecommitValue$) that react to the reception of sufficient enough precommits of a certain type if it has the smallest index (*i.e.*, $p == 0$) or its preceding process with an index $p-1$ has proceeded to the next round already (*i.e.*, $rounds[p-1] > rounds[p]$).

This essentially translates to an observation that must universally holds true across every trace generated by this model for verification:

- Given a round and any upon rule that brings the current step of a process into some later step $s \in \{PREVOTE, PRECOMMIT\}$ denoted $rule_{\{s\}}$, a process with a smaller index must always perform $rule_{\{s\}}$ before any other process with a larger index is allowed to, suggesting that if these actions are chronologically ordered in a timeline forming an execution sequence S, then a filtering of S with respect to the rule type, namely either $rule_{PREVOTE}$ or $rule_{PRECOMMIT}$, followed by a mapping of them into indices of their performing entities, will always culminate in process indices sorted in ascending order, or formally $\forall ruleType \in \{rule_{PREVOTE}, rule_{PRECOMMIT}\} : isSorted(map(processId, filter(ruleType, S)))$, because any upon rule execution that brings the current step of a process with an index p_1 into some later step $s \in \{PREVOTE, PRECOMMIT\}$ can precede the execution of

the upon rule that similarly brings the current step of another process with an index p_2 to a step s if and only if $p_1 < p_2$; and
- Given a round and an upon rule $rule_{NEXTROUND}$ that advances the current round of a process into the next round, a process with a smaller index must always perform $rule_{NEXTROUND}$ before any other process with a larger index is allowed to, indicating that if these actions are chronologically ordered in a timeline forming an execution sequence S, then a mapping of them into indices of their performing entities, will always culminate in process indices sorted in ascending order, or formally $isSorted(map(processId, filter(rule_{NEXTROUND}, S)))$, because any upon rule execution that advances the current round r_1 of a process with an index p_1 into a new round $r_1 + 1$ can precede the execution of the upon rule that similarly advances the current around $r_2 \leq r_1$ of another process with an index p_2 into $r_2 \leq r_1 + 1$ if and only if $p_1 < p_2$. As a direct consequence, when actions performed by different processes at different rounds are interleaved and then chronologically ordered in a timeline forming a global execution sequence S, all actions performed by a process with an index p at a round r must be preceded by the round advancement action performed by a process with an index $p - 1$ at round $r - 1$, or formally $\forall action_1 \in S : \forall action_2 \in S : ((processId(action_1) > processId(action_2)) \land (timestamp(action_1) < timestamp(action_2)) \land (type(action_2) == rule_{NEXTROUND})) \Rightarrow ((round(action_1) < round(action_2)) \land (type(action_1) \neq rule_{NEXTROUND}))$.

Given these two observations articulated, we are now well equipped to prove the correctness of the symmetry reduction applied to this model, or more specifically, to reason about why those execution sequences banned by us and thus omitted from the state space for exploration, do not adversely impact the correctness of the verification outcomes. Instead of attempting to justify the completeness of the reduced state space, we approach it from a different angle by demonstrating that none of the omitted execution sequences contains a state that forms its own orbit, or in other words, belongs to a singleton equivalence relation only comprised of itself, which then deductively proves the bisimulation equivalence between our reduced model (*i.e.*, not a quotient model though due to the non-minimal structure of its constituents) and the ideal quotient model that in theory exists despite the fact that in practice its generation may not necessarily terminate within a reasonable time frame. The high-level proof sketch is outlined below.

- The two axes of symmetry, namely the data values stored in the variables *round* and *step*, with respect to which those two constraints as discussed above are defined, in combination impose a strict total ordering on the execution of each class of upon rules, resulting in omission of any execution of a given class of upon rule (*i.e.*, either $rule_{PREVOTE}$ or $rule_{PRECOMMIT}$) in non-ascending order of process indices, the transformation from which to an symmetric execution sequence in our reduced state space can be proven by construction as follows:
  - Given an arbitrary execution sequence S omitted from the original state space, S must be a valid sequence for exploration, meaning that the order

in which different classes of upon rule are executed by each process must follow a certain sequence as how they are programmed (*i.e.*, $rule_{PREVOTE}$ followed by $rule_{PRECOMMIT}$ and $rule_{NEXTROUND}$);

- Assuming four process indices ranked in (p_1, p_2, p_3, p_4) where p_1 is the smallest and p_4 is the biggest, regardless of the execution order EO of each class of upon rules in S initially, for example $EO = s_0 \xrightarrow{rule_{p_3}} s_1 \xrightarrow{rule_{p_4}} s_2 \xrightarrow{rule_{p_2}} s_3 \xrightarrow{rule_{p_1}} s_4$, we propose a permutation that swaps the process index p_1 of the performing entity with whatever process index of the performing entity that comes first in EO (*e.g.*, $rule_{p_1}$ and $rule_{p_3}$ in the example above), followed by a swap between the process index p_2 of the performing entity and whatever process index of the performing entity that comes second in EO (*e.g.*, $rule_{p_2}$ and $rule_{p_4}$ in the example above), followed by another swap between the process index p_3 of the performing entity and whatever process index of the performing entity that comes third in EO (*e.g.*, $rule_{p_3}$ and $rule_{p_4}$ in the example above), ending up with $EO' = s_0 \xrightarrow{rule_{p_1}} s'_1 \xrightarrow{rule_{p_2}} s'_2 \xrightarrow{rule_{p_3}} s'_3 \xrightarrow{rule_{p_4}} s'_4$ that must always exist in our reduced state space as the only allowable execution sequence for a given rule type, where in this example, the only difference between each pair of states in these two sequences EO and EO' lies in the swapped values of state variables of each pair of swapped performing entities (*e.g.*, (p_1, p_3) for (s_1, s'_1) pair, (p_2, p_4) for (s_2, s'_2) pair, (p_3, p_2) for (s_3, s'_3) pair, and (p_4, p_1) for (s_4, s'_4) pair, resembling the connections between those execution sequences in the generalized diagram presented in Figure 5;
- The proposed permutation strategy must always function as expected, in light of the fact that inherently orderable numerical integers are employed here for indexing processes, making them always sortable in ascending or descending sequence;
- Even though the application of this permutation strategy to both classes $rule_{PREVOTE}$ and $rule_{PRECOMMIT}$ should result in the exact same swapping order of process indices, based on the observation guaranteed by design where the relative order of upon rule executions must be maintained across these two classes of upon rules, or formally $map(processId, filter(rule_{PREVOTE}, S)) == map(processId, filter(rule_{PRECOMMIT}, S))$, indicating that in essence, a *process-based symmetry* is exploited here to enforce the transformed sequence's adherence to an allowable execution sequence from our reduced state space, fulfilling the first observation noted above, yet the same swapping order of process indices applied to the class $rule_{NEXTROUND}$ may not necessarily render it sorted in ascending sequence, and consequently, may not yield a valid execution sequence in our reduced state space, suggesting that a fine-tuned permutation strategy independent of the one employed above for $rule_{PREVOTE}$ and $rule_{PRECOMMIT}$ must be adopted to also mandate an ascendingly sorted indices of performing entities of $rule_{NEXTROUND}$ as well as all the rules dependent on this

round advancement operation, namely all of those upon rule executions in the subsequent round, since, by design, a process cannot execute its actions in a later round without first advancing to that round, and any action performed by a larger-index process must be preceded by its corresponding round advancement operation, meaning that for an action performed by a process with a larger index to precede another, it must have already executed the $rule_{NEXTROUND}$ action, requiring us to permute all actions dependent on $rule_{NEXTROUND}$ performed by a process with a larger index with those dependent on $rule_{NEXTROUND}$ performed by another process with a smaller index, to derive an execution sequence that fulfills the second observation stated above.

- Considering that the same sequence of actions is shared by any execution sequence and its transformed sequence following the aforementioned permutation strategy, with the sole difference being the indices of the performing processes, it follows that corresponding states along both traces must also share identical labelings concerning our LTL formulas on *safety* and *liveness* properties, whose evaluations, by design, can only transition from *false* to *true* but not vice versa, as triggered by the execution of the class $rule_{PRECOMMIT}$.

– As every trace in the full state space can be transformed into an equivalent trace in the reduced state space (as proven above), and vice versa by simply applying the same permutation in a reverse order, implying the absence of any singleton orbit or equivalence class in the set of states involved in the omitted trace, which indicates that every constituent of our reduced model must be a superset of that of the quotient model, or in other words, our symmetry-reduced transition system and the quotient transition system can simulate each other's transitions without exhibiting any observable difference in their sequences of actions with respect to those properties expressed in LTL formulas under verification, we thus have established the bisimulation equivalence between our symmetry-reduced model and the quotient model as a result.
– As per Theorem 1, given the *safety*/*liveness* property expressed in an LTL formula ϕ and the transition system that represents the Tendermint consensus protocol, modelled as a Kripke structure $\mathcal{K}$, by combining $\mathcal{K} \models \phi$ iff $\mathcal{K}_G \models \phi$ implied by the bisimulation equivalence between $\mathcal{K}$ and its quotient model $\mathcal{K}_G$ with respect to an invariance group G and $\mathcal{K}_G \models \phi$ iff $\mathcal{K}_S R \models \phi$ implied by the bisimulation equivalence between the quotient model $\mathcal{K}_G$ and our process-based-symmetry-reduced model $\mathcal{K}_S R$, we get $\mathcal{K} \models \phi$ iff $\mathcal{K}_S R \models \phi$, signifying that model checking our reduced model should reveal all counterexample traces that must also exist in the original full model, and therefore we have proven the validity and correctness of our symmetry reduction method applied here.

References

1. Clarke, E.M., Emerson, E.A., Jha, S., Sistla, A.P.: Symmetry reductions in model checking. In: Computer Aided Verification, 10th International Conference, CAV '98. Lecture Notes in Computer Science, vol. 1427, pp. 147–158. Springer (1998)
2. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.A.: State space reduction using partial order techniques. Int. J. Softw. Tools Technol. Transf. **2**(3), 279–287 (1999)
3. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer (2018)
4. Clarke, E.M., Jha, S., Enders, R., Filkorn, T.: Exploiting symmetry in temporal logic model checking. Formal Methods Syst. Des. **9**(1/2), 77–104 (1996)
5. Godefroid, P.: Exploiting symmetry when model-checking software. In: Formal Methods for Protocol Engineering and Distributed Systems, FORTE XII / PSTV XIX'99, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX). IFIP Conference Proceedings, vol. 156, pp. 257–275. Kluwer (1999)
6. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)

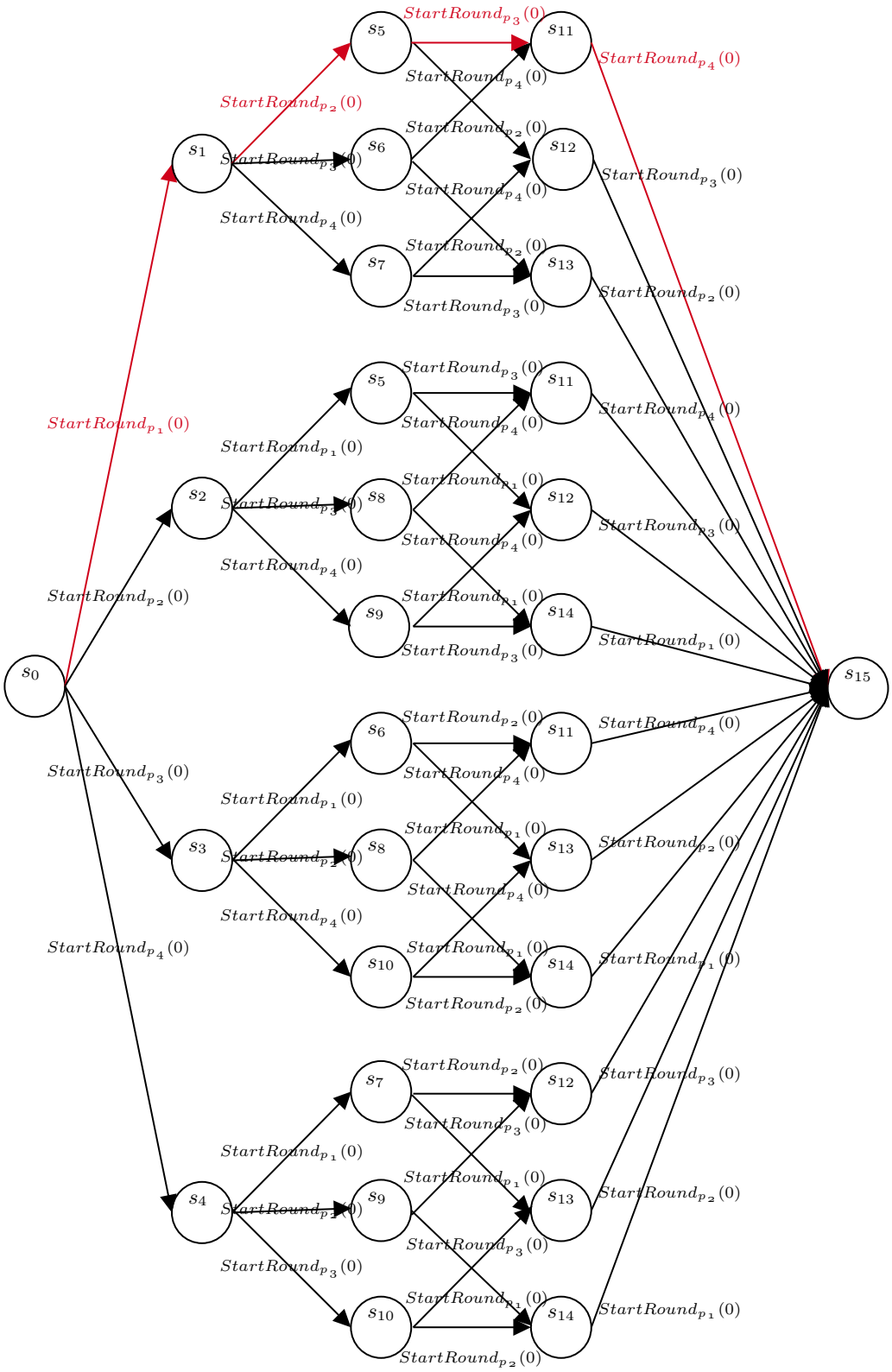

Fig. 3: Partial Order Reduction (Process Level)

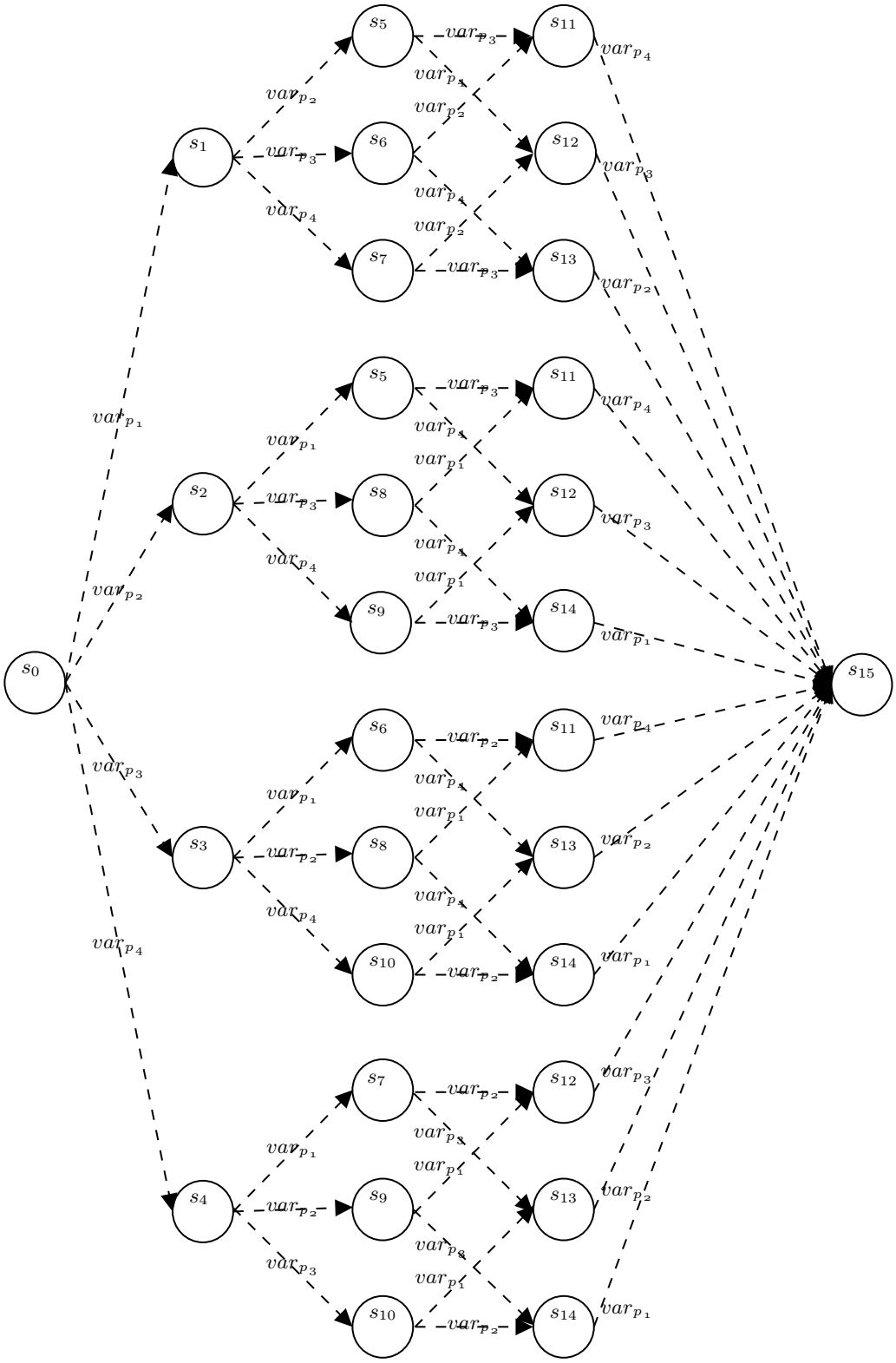

Fig. 4: Partial Order Reduction (Event Level)

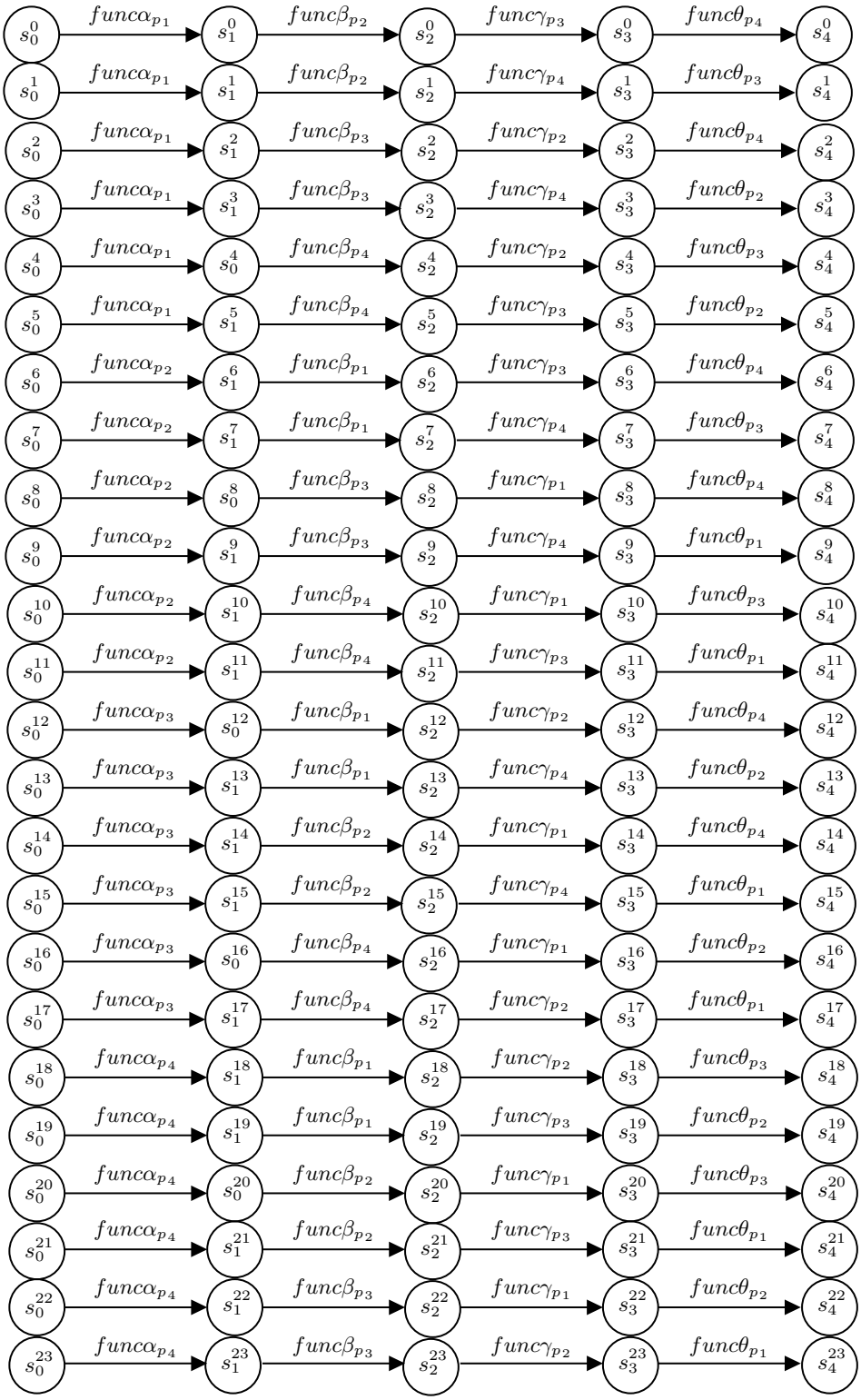

Fig. 5: Process-based Symmetry Reduction