VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
SOFTWARE ENGINEERING STUDY PROGRAM

Optimizing TLA$^+$ specifications for model checking

TLA$^+$ specifikacijų optimizavimas modelių tikrinimui

Master’s thesis

Author:	Marijus Jasinskas	(signature)
Supervisor:	assoc. prof. dr. Karolis Petrauskas	(signature)
Reviewer:	prof. dr. Linas Laibinis	(signature)

Vilnius – 2024

Many thanks to my supervisor, assoc. prof. dr. Karolis Petrauskas, for providing essential guidance, knowledge, and feedback throughout this research project.

I am also thankful to my reviewer, prof. dr. Linas Laibinis, whose remarks helped me to improve this master's thesis considerably.

Summary

In this thesis, a mitigation of the state space explosion problem is presented in the form of TLA$^+$ specification writing guidelines that can reduce the state space size when applied to TLA$^+$ specifications. These guidelines are defined based on the decisions made when writing TLA$^+$ specifications of the Ben-Or distributed consensus algorithm, which has been an important part of research related to the verification of randomized consensus algorithms [BKL$^+$21; KLS$^+$23].

Distributed algorithm TLA$^+$ specification writing decisions relate to the conclusions made in the literature review and a paper in which specification writing guidelines for the mCRL2 language are described [GKO15]. Presented guideline state space reduction effectiveness is measured by comparing the total state space sizes of respective written Ben-Or TLA$^+$ specifications.

Keywords: TLA$^+$, TLC, Ben-Or algorithm, formal methods, specification writing guidelines, state space reduction, state space explosion

Santrauka

Šiame darbe pateikiamos TLA$^+$ specifikacijų rašymo gairės, kurių pritaikymas TLA$^+$ specifikacijose gali sumažinti būsenų erdvės dydį ir taip sušvelninti būsenų erdvės sprogimo problemos padarinius. Gairės yra apibrėžtos pagal sprendimus, priimtus rašant išskirstyto Ben-Or konsensuso algoritmo TLA$^+$ specifikacijas. Ben-Or algoritmas yra svarbus tyrinėjant randomizuoto konsensuso algoritmų verifikaciją [BKL+21; KLS+23].

TLA$^+$ specifikacijose priimti sprendimai siejasi su atliktos literatūros apžvalgos išvadomis bei straipsniu, kuris apibrėžia specifikacijų rašymo gaires mCRL2 kalbai [GKO15]. Pateiktų gairių būsenų erdvės redukcijos efektyvumas yra nustatomas palyginant atitinkamų sukurtų Ben-Or TLA$^+$ specifikacijų pilnos būsenų erdvės dydžius.

Raktiniai žodžiai: TLA$^+$, TLC, Ben-Or algoritmas, formalūs metodai, specifikacijų rašymo gairės, būsenų erdvės redukcija, būsenų erdvės sprogimas

TABLE OF CONTENTS

Introduction

Most software systems are implemented based on a design specification that encompasses system architecture, components, modules, interfaces, data, and information [Nic11]. As the capabilities of humankind to create innovative software solutions that solve urgent problems increases, so does the complexity of those solutions due to the ever-growing needs of modern society in terms of system availability, scalability, and performance [EMB24; KMM18].

Systems become harder to design, maintain, and adapt to environments of rapidly evolving technologies such as artificial intelligence [Rus24]. That is why in a modern environment it is paramount to have a correct specification as a basis for a system's implementation and to verify and validate both the design and written code to ensure that requirements for the system are met [ZZW$^+$24].

This is crucial for distributed systems that support important infrastructure: telecommunication networks, the World Wide Web, aircraft control systems, and various other safety-critical systems [GvdPW23; KDL$^+$22]. The issue is that it is more difficult to identify defects in big complex distributed systems. Rigorous testing and quality assurance can only help up to a certain point since various unexpected scenarios could occur more frequently when many different computers communicate with each other over separate networks [AZT$^+$24; FGH22; GvdPW23].

Formal methods can alleviate the effects of this problem in the form of modeling, analyzing, and verifying the software system design [GM12]. If it is possible to define the intended behavior of a distributed system formally, then this definition can be further examined with tools to verify certain required properties [GvdPW23].

However, creating a formal definition of system behavior poses its own challenges. To describe a software system with formal syntax and semantics while making sure that this description abides by factors present in the real world is not an easy task [GvdPW23; Pro24]. Over-abstraction of software behavior leads to formal definitions that might not be very useful unless a refinement that includes more details related to the implementation of the system is also produced [GM12].

Furthermore, the adoption of formal methods by software developers is not straightforward as many formal method tools and formal specification languages with their specific syntax, semantics, and other complexities exist [GM20; GtBvdP20]. Given that some of these languages and tools are less popular and less documented than other forms of verification (mainly, software testing), it is difficult to integrate their use into the software development process in the industry [GvdPW23].

Despite these challenges, some formal method tools and languages are becoming more relevant due to their effectiveness in helping to identify unforeseen defects, especially in large-scale distributed systems [GvdPW23]. At the time of writing, a great example of this is the TLA$^+$ (Temporal Logic of Actions) formal specification language.

TLA$^+$ is used to describe state machines that model system behavior with temporal logic formulas [Lam02]. It is based on the Zermelo-Fraenkel set theory and Temporal Logic of Actions (TLA) which is a variant of linear-time temporal logic [KKM22].

TLA$^+$ has already been actively used by big tech companies like Amazon, Intel, and Microsoft [Bee08; NRZ$^+$15]. It is mostly used for modeling large-scale concurrent and distributed systems or safety-critical systems since it allows to understand potential unwanted behaviors in the design of such systems [New14; VBF$^+$11].

Recently, the Linux Foundation announced the creation of the TLA$^+$ Foundation with Microsoft, Oracle, and Amazon being its inaugural members [Fou23]. This means that TLA$^+$ could become adopted by more companies and potentially become more popular within the software development industry.

A TLA$^+$ specification describes states as an assignment of values to defined variables [Lam02]. Each variable var has a value at the current state and the values of each variable at the next state are defined as an assignment of a new value to var'. If some variable var does not change its value in the next state, the *UNCHANGED* operator can be used to explicitly define what variables' values do not change [KKM22].

Propositional logic operators ($\wedge$, $\vee$, $\implies$, and others) are used to define formulas for the possible behaviors of the state machine model [Lam02]. Temporal operators $\Box$ (always) and $\Diamond$ (eventually) are used for checking invariants and temporal properties of the system's behavior.

TLA$^+$ uses TLC to check all possible states and whether the properties defined in the specification are true for every state in the model [KKM22]. The problem is that the TLC model checker uses a brute-force algorithm for checking states [Lam02].

When the number of states in the system grows, the amount of time required to check the whole state space with TLC increases exponentially [OKK$^+$23]. This is known as the state space explosion problem and it has been encountered as early as the second half of the 1980s [LCL87]. Since then many state space explosion relief strategies and state space reduction methods have been proposed and discussed among the scientific community [CGJ+01; EGK+13; GKO15; NGY+21; Pel09; Val98].

For TLA$^+$, the main focus of work done to solve the state space explosion problem with TLC is related to improving the model checker itself [OKK+23]. A symbolic model checker Apalache for TLA$^+$ is currently being developed to reduce the amount of time required for verifying temporal properties defined in TLA$^+$ specifications [Apa24; KKT19].

While improvements made to the model checker help to mitigate the state space explosion problem, not as much attention is given to different styles of writing TLA$^+$ specifications so that they would be easier to verify.

If some ways of describing system behavior in TLA$^+$ specifications lead to a smaller state space size of a state machine model in certain cases, then these writing patterns should be defined and evaluated in terms of their state space reduction effectiveness, advantages, and drawbacks for them to potentially be used by TLA$^+$ specifications authors.

The aim of this master's thesis is to define TLA$^+$ specification writing guidelines that when applied would allow to reduce the state or behavior space size of a state machine model defined with TLA$^+$.

These guidelines are intended to be applied when modeling distributed systems with TLA$^+$. Information that explains in what circumstances these guidelines can be applied and how their state space reduction and effectiveness were shown is provided in the thesis as well.

It is hypothesized that the defined specification writing guidelines applied to TLA$^+$ specifications correctly would reduce the state or behavior space size of state machine models formulated with TLA$^+$. For correct application of a guideline to a TLA$^+$ specification, it must be applied within the provided context – the required application properties that are presented in the definition of that guideline must be true.

The expected results of the master's thesis are TLA$^+$ specification writing guideline definitions with their proofs of effectiveness and correctness, and a description of an algorithm that would detect whether a guideline could be applied to a TLA$^+$ specification and if that specific guideline was applied correctly.

1. State of the art

This section contains a conducted literature review to gather the required knowledge necessary to achieve the aim of the master's thesis. The literature review focuses on the current methods of state space reduction that are used effectively on TLA$^+$ specifications. All the ways to reduce state space size in TLA$^+$ are presented from the perspective of distributed algorithms.

The goal of this literature review is to identify how TLA$^+$ specifications could be written to reduce state space size and how that impacts the state machine model's representation of the real-world system behavior. This would later allow to formulate potential TLA$^+$ specification writing guidelines, their application cases, and trade-offs.

The literature review was done based on the concepts of the snowballing method defined in [Woh14]. Backward and forward snowballing was used to find new papers to expand the set of papers reviewed.

This literature review is also a state of the art review since the latest scientific papers and conference material were considered when looking for information to review. It describes how state space reduction methods in the context of TLA$^+$ have improved over the years and what has been accomplished so far.

1.1. Why state space increases

To better understand how to decrease the state space size of a TLA$^+$ specification, it is helpful to know what could be avoided when writing TLA$^+$ specifications. The circumstances mentioned in this subsection are the most common causes of increased state space and could sometimes be circumvented based on what distributed algorithm is being modeled.

1.1.1. Unnecessary variables

The most obvious way to increase the state space size of any TLA$^+$ specification is to add more variables. By defining more variables, the number of possible behaviors increases since there are more possible sets of variables and their values.

The state space grows exponentially because every new variable added also has relationships with other variables. Since TLC checks all possible executions, all possible orders in which variable values are assigned will also be checked [Lam02]. This means that if a variable that is a set with several elements is added to a specification, it can have a huge impact on the growth of the size of the state space.

1.1.2. Records

Most of the time records are used to model sent and received messages in various distributed algorithms. They increase the state space size of a TLA$^+$ specification considerably since every record field can have more than one possible value and that field could be an element of a set or a whole set.

However, to make sure that the general behavior of a distributed algorithm is modeled explicitly enough to check invariants and temporal properties, records are necessary most of the time. An example of this would be how the Paxos consensus algorithm's [Lam01] TLA$^+$ specification uses records as messages that are added to a set of all messages ever sent [Git20]. Messages have the acceptor field to make sure that several messages with the same values can still be included in the set. Since sets can not contain duplicate elements, a bag could be used instead and fields that describe the message sender could be omitted.

In the general case, if all sent messages are added to the same set, a field for the sender of every message is added to a record to allow to count the number of messages received based on the cardinality of a subset of all messages.

The Raft consensus algorithm's [OO14] TLA$^+$ specification also uses records to represent messages but instead adds them to a domain of a function that maps a certain record to the number of times it was sent [Git18].

Records might be unnecessary for simpler distributed algorithms or they could be omitted in higher abstraction level TLA$^+$ specifications of complex algorithms.

1.1.3. CHOOSE and the existential quantifier

The existential quantifier $\exists$ can increase the state space size of a TLA$^+$ specification considerably compared to the *CHOOSE* operator. There is a significant difference between $\exists$ and *CHOOSE* in TLA$^+$. When the *CHOOSE* operator is used, a single value that satisfies a defined predicate is picked and used for all executions of the model by TLC [Lam02]. This means that if there is a need to check behaviors with all possible values that satisfy a predicate, *CHOOSE* should not be used.

The existential quantifier $\exists$ allows checking all possible values that satisfy a predicate with TLC [Lam02]. But this can have a huge impact on the state space size of the specification. The impact on state space growth depends not only on the number of possible choices covered by the existential quantifier but also on the number of times the existential quantifier is used in the specification.

Both of these operators have use cases where they can be utilized most effectively. The Raft consensus algorithm's TLA$^+$ specification utilizes *CHOOSE* to return the minimum and maximum value elements of a given set [Git18]. This is a valid use of the operator because there can only be one possible minimum or maximum value in a finite set. The existential quantifier $\exists$ is used in the next state formula $Next$ to allow for any node to perform any of the appropriate actions defined in the algorithm.

In the Paxos consensus algorithm *CHOOSE* is used to obtain some random value that is not in a defined set of values [Git20]. Since TLC will always be given the same value for all executions with *CHOOSE*, in this particular case *CHOOSE* is used to obtain a kind of *NULL* or *Undefined* value. The existential quantifier $\exists$ in the Paxos specification is used similarly to the Raft specification.

In most distributed algorithms, $\exists$ can not be omitted due to the need for TLC to check all possible behaviors and confirm whether invariants or temporal properties hold for all possible cases.

1.1.4. Lenient state conditions

State space can increase in TLA$^+$ specifications when conditional operators are used too leniently to define what the next possible states are. Actions that could occur from the same state are usually written with the disjunction propositional logic operator $\vee$.

An action formula $A \vee B$ implies that both actions A and B can occur in the next state [Lam02]. In TLA$^+$ specifications it is essential to ensure whether it is intended and correct for actions A and B to both possibly occur from the same state. If A and B do occur from the same state, two or more new behaviors are evaluated by TLC.

Disjunction can still be used in cases where only one action A or B can occur in the next step.

Additional conditions for actions A and B could be added with the conjunction operator to ensure A and B never both occur from the same state.

$$(A \land \neg C) \lor (B \land C) \tag{1}$$

(1) defines a step in which only one action A or B could occur, depending on the value of the formula C. Another alternative to making sure that actions A and B could never both occur from the same state is to use the *IF THEN ELSE* operator.

$$\text{IF } C \text{ THEN } B \text{ ELSE } A \tag{2}$$

$$\text{CHOOSE } V \, : \, (C \implies (V = B)) \land (\neg C \implies (V = A)) \tag{3}$$

(1) and (2) both allow for only one action A or B to occur in the next state of the system. The definitions presented in (2) and (3) are equivalent [Lam02].

Sometimes it is forgotten to check what level of leniency is required for certain actions in a specification. The number of states and behaviors in a TLA$^+$ specification grows the more lenient conditions are formulated for the next state's actions. Not only because there are more possible actions that could occur in the next state, but also because there are more possible orders in which actions are taken in different behaviors.

1.2. State space explosion relief strategies

The strategies mentioned in this section do not impact the state space size of a TLA$^+$ specification directly. Instead, they allow to avoid the state space explosion problem to some extent.

1.2.1. Using TLAPS

An alternative to using the TLC model checker to check all states is to use the TLAPS (TLA$^+$ Proof System) to formally prove that certain invariants are true [CDL$^+$12]. This is done by constructing mathematical proofs and using mathematical solvers to formally prove theorems for a particular TLA$^+$ specification. TLAPS does not require traversing the whole state space and is faster than TLC when there are a lot of possible states and behaviors.

However, the construction of mathematical proofs, especially for proving inductive invariants, is a challenging task that requires a considerable amount of time and effort [CDL+12]. This can be mitigated by using mathematical proof techniques [Vel19] and defined keywords for proof simplification (*HAVE*, *TAKE*, *WITNESS*, *PICK*, and *SUFFICES*) [Inr24a].

Sometimes, certain theorems can not be proven without defining specific assumptions for the specification. Assumptions are written with the *ASSUME* keyword and are based on conditions that are upfront requirements for a given distributed algorithm [Lam02]. For example, this could be the assumption that the number of faulty nodes f has to be lower than a certain threshold value of the number of all nodes N.

TLAPS does not automatically expand formula definitions when proving theorems [KKM22]. These have to be manually expanded with the *DEF* keyword. Very long and complex formulas have to be divided into smaller parts and proven separately from one another.

Another issue is that if the TLA$^+$ specification changes, TLAPS proofs would have to be modified accordingly. This is not ideal when complex proofs get long and they have to be readjusted.

If certain properties can not be proven with TLAPS formally, errors might be present in the TLA$^+$ specification. They can be identified without using TLC by writing test case proof steps with *PROOF OMITTED* and examining what proofs are still missing and can not be proven without omission. Although, in such cases, it is recommended to consider running TLC on a specification with a smaller state space size [Inr24b]. This could help identify potential errors more reliably than examining how proof steps are evaluated.

There are also general theorems based on set theory, functions, and other fields of mathematics defined and provided with a TLAPS installation [KKM22]. These theorems must at times be explicitly used to prove a safety or liveness property of a specification with the *BY* keyword.

A very important part of working with TLAPS is making sure to use the correct backend solvers for proofs [Inr24b]. One of the most frequently used TLAPS solvers is the propositional temporal logic (PTL) solver that allows proving inductive invariants such as *TypeOK* [KKM22].

1.2.2. Refinement

Specification refinement can be used to mitigate the state space explosion problem and allow to prove certain temporal properties with the TLC model checker quicker.

If there is a specification B and it has a huge state space size, a separate specification A

that has fewer states could be written. Then there could exist such a refinement mapping R that would map all the possible states and behaviors of specification A to a subset of possible states and behaviors of specification B.

If such a refinement mapping R is proven to be true with TLAPS or TLC, then all behaviors that are valid in specification A would also be valid in specification B. This would mean that all temporal properties that are true for specification B would also hold for specification A [LM22]. Due to the smaller state space size of A, temporal properties of the specification could be checked with TLC much more quickly.

However, checking temporal properties only in specification A might not be enough to verify them in specification B. It would be possible to define such a refinement mapping R that would map all valid behaviors of specification A to the initial state in specification B. Then valid temporal properties in specification A would not necessarily show whether such temporal properties are true in all possible behaviors of specification B.

Additionally, defining and proving refinement mappings usually requires some additional variables called auxiliary variables [LM22]. They are used to store certain information that would enable the mapping of certain behaviors of different specifications.

There are three main kinds of auxiliary variables used for refinement mappings [LM22]. History variables store information about the previous and current states. Prophecy variables are used to predict what the next state action will be. Stuttering variables add additional stuttering steps so that both specifications have the same number of stuttering steps for certain behaviors. All of these require additional analysis of both specifications to be utilized effectively and what kind of auxiliary variable to use depends on how specifications A and B differ from one another.

1.3. Model checking optimization methods

The methods described in this section encompass the model checking process and its effectiveness. Various techniques can be used to reduce the state space size of a specification during model checking. Of course, a particular model checker has to support the implementation of such methods.

1.3.1. Partial order reduction

Partial order reduction aims to minimize the number of modeled orders of states in system behaviors to reduce the state space size for model checking [BK08]. This is done by identifying concurrent state transitions whose effects are independent of their order of execution.

Let us define an operator for sending a broadcast message by adding a record to a set of all sent messages.

$$\begin{aligned} &Send(node) \triangleq \\ &\quad msgs' = msgs \cup \{[type \mapsto \text{“}request\text{”}, value \mapsto proposal[node], sender \mapsto node]\} \end{aligned} \tag{4}$$

The operator $Send$ defined in (4) models the sending of a broadcast request message with a certain value. This value is the node's current proposal and is used to reach a consensus among all nodes in a distributed system. The sender field is used to make sure that several messages of the same type and value can be included in the set of all sent messages.

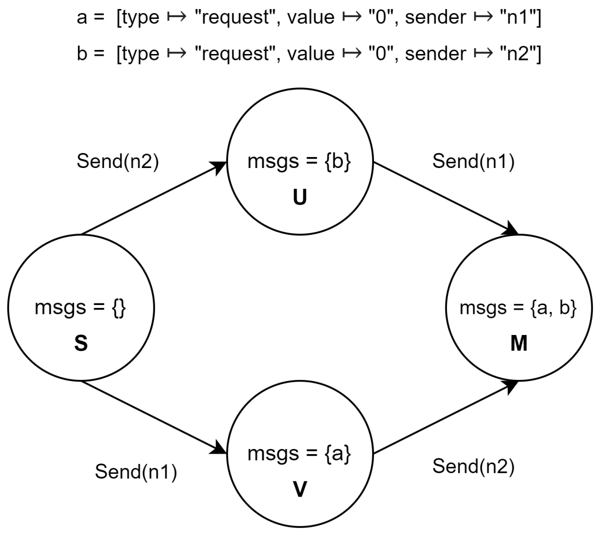
a = [type ↦ "request", value ↦ "0", sender ↦ "n1"]
b = [type ↦ "request", value ↦ "0", sender ↦ "n2"]
Send(n2)
msgs = {b}
U
Send(n1)
msgs = {}
S
msgs = {a, b}
M
Send(n1)
msgs = {a}
V
Send(n2)

Figure 1. Partial order reduction for a distributed algorithm

Figure 1 depicts the states and behaviors of a distributed algorithm where node $n1$ sends a message a and node $n2$ sends a message b with the $Send$ operator defined in (4).

The possible states of the system are S, U, V, and M, and the possible behaviors of the system are $S \rightarrow U \rightarrow M$ and $S \rightarrow V \rightarrow M$. If messages a and b have the same value and only differ by their sending node, and if both nodes $n1$ and $n2$ are of the same role (voters) then, in general, it should not matter whether $n1$ or $n2$ sends their respective message first. In that case,

$S \to U \to M$ and $S \to V \to M$ are independent behaviors and they would lead to the same state M. With partial order reduction, one of these behaviors could be entirely omitted during model checking, and the state space that needs to be traversed by the model checker would be reduced.

The types of messages a and b, and types of nodes $n1$ and $n2$ matter since, depending on the distributed algorithm, the order of message sending could result in different states and could not be reduced. It is also important to mention that states U and V must transition to only state M. This means that, in this particular case, the addition of one message to the set $msgs$ does not cause any additional state transitions to occur.

There are a couple of variants of partial order reduction. Static partial order reduction as defined in [KLM+98] computes the reduced state space before checking the whole state space of the system model. Dynamic partial order reduction [FG05] does certain computations before model checking but mostly calculates the reduced state space during the process dynamically. On large system models, dynamic partial order reduction performs worse than static partial order reduction [Akh12].

TLC does not support partial order reduction and instead uses a brute-force breadth-first search algorithm for state space traversal [YML99]. There has been a proposed partial order reduction implementation for a modified TLC model checker [Akh12]. However, this project has been discontinued and, at the time of writing, TLC still does not support any form of partial order reduction [Gro24].

1.3.2. Symmetry sets

A set S is a symmetry set if and only if permuting its elements in any way does not change whether or not a behavior is valid in a given TLA$^+$ specification. In TLA$^+$ sets can be declared as symmetry sets in the configuration file with the *SYMMETRY* keyword [Lam02]. Once a set is declared a symmetry set, TLC will only check behaviors with one chosen permutation of that set's elements.

N_1 = {"n1", "n2", "n3"} N_2 = {"n2", "n1", "n3"} N_3 = {"n3", "n2", "n1"}

N_4 = {"n1", "n3", "n2"} N_5 = {"n2", "n3", "n1"} N_6 = {"n3", "n1", "n2"}

Figure 2. Sets N_1 to N_6 are all equivalent, N_1 is a symmetry set

Figure 2 shows a symmetry set N_1 that contains node identity values that are commonly used in most distributed algorithm TLA$^+$ specifications. Since the cardinality of the N_1 set is 3, the total number of omitted executions with symmetry by TLC is $3! - 1$. If there are a lot more symmetry set elements, the state space size reduction becomes greater.

There are certain difficulties with using symmetry sets. Even though it is possible to declare several separate sets as symmetry sets in a configuration file, is it required that the union of all symmetry sets can not contain elements of different types.

Furthermore, TLC does not check whether a set declared with a *SYMMETRY* keyword is really a symmetry set. This means that it is possible to remove important states and behaviors from model checking if the set is not a symmetry set.

Most importantly, using symmetry sets is only viable when trying to prove safety properties, as liveness property verification is not supported with symmetry sets [Lam02]. Additionally, if the symmetry set is very large, TLC will take some time to remove other possible orderings of set elements and can take longer to check the model on specifications with a small state space size.

1.3.3. Simulation mode

TLC can also be run in simulation mode which randomly samples the state space instead of checking all possible states and behaviors up to a certain depth of the constructed state graph [Lam02].

This approach does not produce a full verification of the specification, but it can help to detect certain errors without the need for long model checking sessions.

Despite that, the random sampling can lead to missing some design bugs and could end up making it only seem like everything is correct. If the state space is small enough, it would be better to use the normal TLC mode for model checking to avoid false positive assumptions about the validity of the specification.

Each time TLC is run in simulation mode it picks states based on a random generation seed and an *aril* value [Lam02]. Both of these values can then be inputted during the next TLC simulation mode session as additional parameters to reproduce the same choices of states.

Found errors can be investigated more thoroughly by adding *Print* statements to the TLA$^+$ specification. *Print* statements do not affect the outcome of a TLC model checking session in simulation or normal modes [Lam02].

1.3.4. Using symbolic logic and Apalache

Symbolic model checkers can evaluate a huge number of states in one step by representing them as single formulas.

Binary decision diagrams are used to represent boolean formulas and to model check them more effectively [BCM$^+$92]. However, the construction of binary decision diagrams heavily depends on the order of variables in a specification. If the variables are defined in an unfavorable order, the constructed binary decision diagram would be unnecessarily complex and less effective for symbolic model checking. Reduced ordered binary decision diagrams were defined to mitigate this problem [PAH$^+$08]. They are binary decision diagrams with stricter formulation requirements that allow to create a structure with fewer unnecessary elements.

Apalache is a symbolic model checker for TLA$^+$ that utilizes Satisfiability Modulo Theory (SMT) solvers [KKT19]. Currently, it does not fully support all TLA$^+$ operators but there are methods provided by the developers of Apalache to overcome these limitations [Apa24]. This makes Apalache a preferable alternative to the TLC model checker due to its greater effectiveness in model checking [KKT19].

Recently there have been improvements made to Apalache based on SMT array theory that have increased its effectiveness considerably [OKK$^+$23]. Apalache could be modified to also use partial order reduction for even greater model checking effectiveness.

1.4. Proposed specification writing guidelines

[GKO15] describe seven guidelines for reducing state space in state machines defined with mCRL2 (micro Common Representation Language 2) [mCR24] specifications. However, the authors do not provide any data regarding the equivalency of specifications before and after guideline application. This is left to the guideline applier to assess themselves.

It would be more useful if there was some data provided and a way to let readers know what trade-offs certain guidelines entail exactly. It is apparent that guidelines should be applied when modeling certain concepts or processes when their application effect is satisfactory to reach a desired outcome.

At the time of writing, the [GKO15] paper is still relevant and is the only paper that the author was able to find that defines state space reduction guidelines for writing specifications. The seven guidelines defined in [GKO15] are illustrated with examples of traffic light controller systems that give insight on how and when the guidelines should be applied.

A very important aspect of these guidelines is that they are based on a "*design for verifiability*" approach. This means that system models are created in such a way that their behavior could be verified easily. The authors note that there has not been an emphasis on how to create system models that are more easily verifiable. Instead, most research has been conducted on how to verify already existing system models more effectively [GKO15].

There is no guarantee that applying a certain guideline defined in [GKO15] will yield a result that has a smaller state space size, is verifiable, and also behaviorally equivalent in all cases. The authors argue that the more traditional approach of state space reduction that preserves equivalences is far less effective than their proposed guidelines.

However, the trade-offs of guidelines defined by the authors should be quite well documented so that specification writers could understand when to apply them. The fact that the authors use only traffic light controller systems to illustrate the applications of their guidelines does not fully show how they would be useful when modeling other kinds of systems. The authors do, however, give recommendations on what type of system models could benefit from the application of each guideline.

For distributed algorithms preserving specification equivalences is not mandatory if there is a certainty that specific safety or liveness properties are preserved. This is because if the specification writer knows what properties need to be proven, they can use a guideline that would allow them to reach the desired outcome.

There might exist such state space reduction guidelines that preserve certain properties only for certain distributed algorithms. And some of these guidelines might in some way be related to the guidelines described in [GKO15]. Of course, the trade-offs and application cases of these guidelines would need to be provided for them to be useful in the modeling of distributed algorithms.

1.4.1. Guideline I: information polling

The first guideline defined in [GKO15] states that specifications should model information polling instead of information pushing because then the size of the state space is smaller. It is mentioned that a system that communicates based on information polling could overload communication networks with an overwhelming number of messages about events being shared between servers and clients.

In most distributed algorithm TLA$^+$ specifications, nodes only check the number of particular messages when they need to make decisions on what messages to send next. This means that all messages that are required are received and evaluated at once.

There are a few considerations when modeling message communication in distributed algorithms. It is possible to explicitly model message content or just have boolean variables that would show whether a certain quorum has been formed like the boolean variables for triggers defined in the polling example shown in the paper [GKO15].

The boolean variable approach would reduce the state space size quite considerably, however, for most distributed algorithms modeling messages explicitly is essential to proving certain safety or liveness properties. Nevertheless, such an approach might be enough to prove that consensus is eventually reached or that no two nodes can become leaders at the same time.

Since messages are omitted entirely they do not have to be stored for later use like the pushed signals in the traffic light controller example for the information polling guideline [GKO15]. Additionally, each node must keep track of its flag values to make sure it is in the correct phase of the algorithm. The sensors provided in the example [GKO15] must also keep track of whether they have been triggered or not.

A system's specification does not necessarily need to model information pushing just because it would be a better implementation option for the real-world system. If the main purpose of a certain specification is to prove the safety or liveness properties of a distributed algorithm, information polling could be modeled. That would allow to prove specific properties with model checking more quickly due to a smaller state space size.

1.4.2. Guideline II: use global synchronous communication

The second guideline defined in [GKO15] states that communication should be modeled synchronously so that messages are received and forwarded instantaneously. This eliminates unnecessary states for communication chains that contain several communicating components because a single message passing several components is modeled as a single behavior. The example given for the guideline is simple and does not have a lot of different behaviors.

In mCRL2 the operators $\cdot$ and $||$ are used for defining whether actions happen sequentially or in parallel. In TLA$^+$ disjunction ($\vee$) is used to describe actions that could occur at the same time in the modeled system.

Distributed algorithm specifications model how each node receives and sends messages separately. There is a set of nodes that has elements that represent nodes (similar to sets depicted in Figure 2). For each node operators for message sending, receiving, and entering various phases of a distributed algorithm are defined as actions that occur in different steps. This is done because it is impossible to determine which subset of nodes will advance to the next phase of the algorithm or reach a consensus.

To prove safety or liveness properties all possible node actions must be examined. What is more, nodes could also fall off and be stuck in different rounds. If faulty nodes are also modeled, then the most important parts of node communication should not be over-abstracted in the specification.

Due to these reasons, distributed algorithm specifications allow for only one node's actions to occur in a single step. An application of the global synchronous communication guideline would mean that when one node receives a message it sends another message and then the node that receives it can send its message in the same step.

1.4.3. Guideline III: avoid parallelism among components

The third guideline defined in [GKO15] states that behavior among parallel components of a system should be sequential to reduce the size of the state space. If parallelism is avoided in specifications, then the state space size grows only linearly in the number of components, not exponentially.

This guideline is not relevant for distributed systems because nodes in a distributed system operate in parallel inherently. Unless certain properties must be proven, there can be no use in making a distributed system sequential since it would not represent the real-world system being modeled at all. The authors in [GKO15] mention that avoiding parallelism does not provide beneficial results when used with symbolic logic model checking.

1.4.4. Guideline IV: confluence

The fourth guideline defined in [GKO15] describes how τ-confluent system state spaces can be reduced with τ-prioritization and branching bisimulation reduction [vGW96]. These techniques also guarantee that the reduced specification would be behaviorally equivalent, and can reduce the size of the state space quite considerably even for complex system specifications.

It is important to note that for this approach to work the following must be true:

1. The system model must be τ-confluent.
    - This can be ensured by examining a behavioral description of the system.
    - It is not necessary to generate the full set of states and behaviors to ensure τ-confluence applies to a system's model.
2. The author of the specification must correctly denote which state transitions are "*empty multi-actions*" [GKO15] which are referred to as τ actions.
    - If τ-confluence is ensured then that means it is known which state transitions are τ actions.
    - mCRL2 supports a hiding operator written as $\tau_{\{\}}$. TLA^{+} does not support such an operator.
    - If TLA^{+} did support such an operator it could be applied to defined action formulas.

The authors note that information pushing prevents a system model from being τ-confluent [GKO15]. However, if the system is τ-confluent, then τ-prioritization can be executed on the system state space to reduce its size by a considerable amount. After that, branching bisimulation reduction can reduce the τ-prioritization's reduced state space even more in most cases.

For distributed algorithms, τ-confluence could potentially be an effective state space reduction method. For example, the transitions of message sending could be omitted as a τ action. If message receiving is also omitted as a τ action, then a specification with a lot fewer states would be produced.

1.4.4.1. Using τ-confluence for state space reduction

It is also important to state that τ-confluence is very similar to partial order reduction [GvdP00]. In essence, τ-confluence and partial order reduction both aim to remove unnecessary state transitions to reduce the state space size of a specification. The major difference is that τ-confluence always preserves branching-time properties [GvdP00].

To illustrate how τ-confluence could be used for state space reduction in distributed algorithm TLA$^+$ specifications let us consider a flawed yet simple distributed consensus algorithm:

1. We assume that the number of nodes $N \geqslant 3$.
2. Every node sends a broadcast request message $R(p)$ that contains a proposal value $p \in \{0, 1\}$.
3. Once a node receives more than $N/2$ request messages $R(p)$ from other nodes, it sets its decision value D to the proposal value $p_{majority}$ that was sent by the majority of nodes.
4. If there are multiple possible values $p_{majority}$, the node sets its decision value D to any one of them: $D = 0 \lor D = 1$.
5. The algorithm terminates once all nodes have decided on some value:

$$\forall\, node \in Nodes : D[node] \in \{0, 1\}.$$

This algorithm is only meant to be an example of how τ-confluence could be used and does not actually guarantee that all nodes will decide on the same value. Such a flawed example is still valuable because this defined algorithm exhibits characteristics that are present in other distributed algorithms.

It requires a certain number of messages of a certain type to be sent and received until the next step of the algorithm can be executed. This means that the order in which messages are sent is meaningful only when it determines the elements of the set of all sent request messages. The TLA$^+$ specification written for the algorithm is provided in Figure 17.

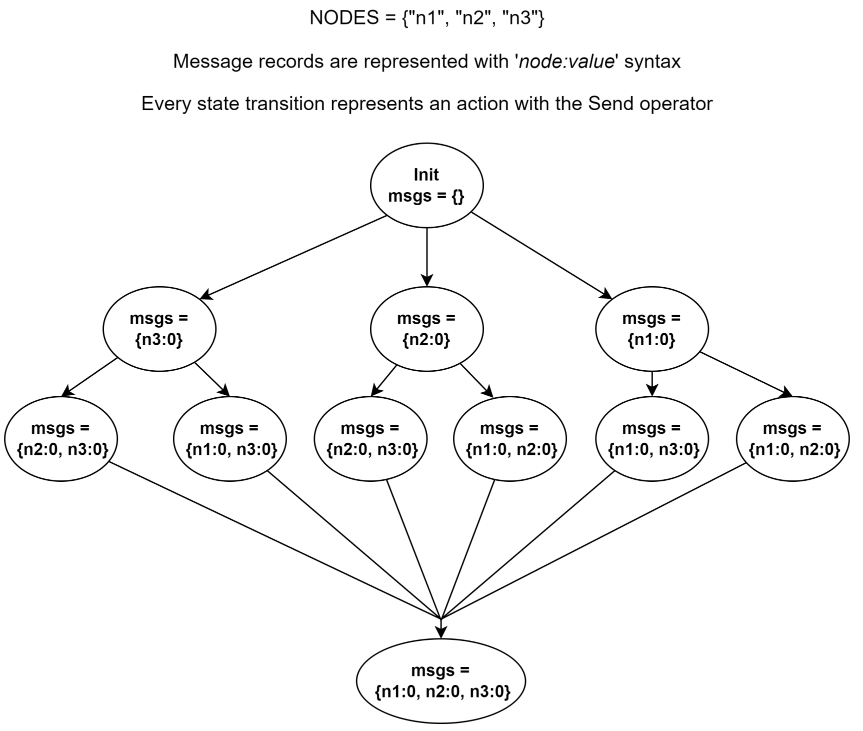
NODES = {"n1", "n2", "n3"}
Message records are represented with '*node:value*' syntax
Every state transition represents an action with the Send operator
Init
msgs = {}
msgs = {n3:0}
msgs = {n2:0}
msgs = {n1:0}
msgs = {n2:0, n3:0}
msgs = {n1:0, n3:0}
msgs = {n2:0, n3:0}
msgs = {n1:0, n2:0}
msgs = {n1:0, n3:0}
msgs = {n1:0, n2:0}
msgs = {n1:0, n2:0, n3:0}

Figure 3. All possible state transitions with state actions

Figure 3 shows all possible state transitions of the algorithm specified in Figure 17 and described in this subsection. The state transitions are steps in which the $Send$ action is taken – analogous to the $Send$ operator defined in (4). Even though there are a lot of possible behaviors that represent orders in which nodes send messages with the same value, some of them could be entirely removed.

Consider a definition of a new TLA^{+} operator in the $Spec$ formula described in Figure 17:

$$Init \land \Box\,[Next]_{vars} \land \tau(Send) \tag{5}$$

The τ operator in (5) could tell a modified TLC model checker that all actions with the *Send* operator are τ actions. Then TLC could check which of the τ actions could be removed while preserving specification equivalence.

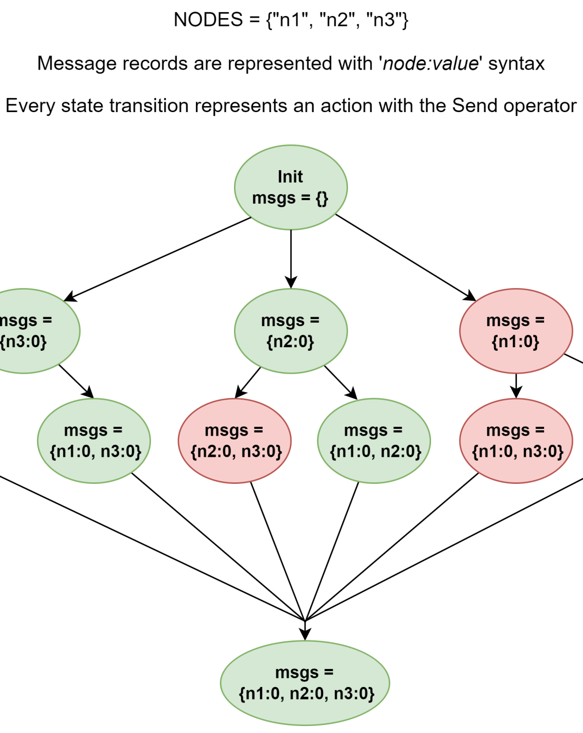
NODES = {"n1", "n2", "n3"}
Message records are represented with '*node:value*' syntax
Every state transition represents an action with the Send operator
Init msgs = {}
msgs = {n3:0}
msgs = {n2:0}
msgs = {n1:0}
msgs = {n2:0, n3:0}
msgs = {n1:0, n3:0}
msgs = {n2:0, n3:0}
msgs = {n1:0, n2:0}
msgs = {n1:0, n3:0}
msgs = {n1:0, n2:0}
msgs = {n1:0, n2:0, n3:0}

Figure 4. Unique and duplicate states marked by green and red respectively

Figure 4 shows how the τ operator and a modified TLC model checker could potentially identify what state transitions could be removed. This would result in the reduced state space of the state machine model shown in Figure 5.

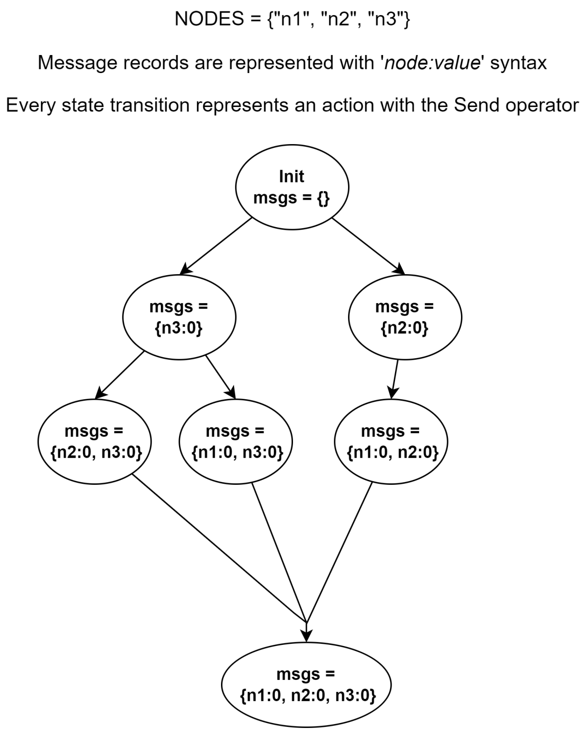
NODES = {"n1", "n2", "n3"}
Message records are represented with '*node:value*' syntax
Every state transition represents an action with the Send operator
Init msgs = {}
msgs = {n3:0}
msgs = {n2:0}
msgs = {n2:0, n3:0}
msgs = {n1:0, n3:0}
msgs = {n1:0, n2:0}
msgs = {n1:0, n2:0, n3:0}

Figure 5. Result of state space reduction with τ-confluence

The states that were omitted in Figure 5 did not have any impact on the possible decision values. This is true because all nodes propose the same value 0 and it does not matter which nodes sent the majority of values. More difficult cases would require careful examination of the state space of a distributed algorithm to find possible actions that could be omitted.

It is important to mention that the $Send$ operator also encompasses state transitions after the required number of messages for making a decision is reached since one more node can always send a message. A separate operator called $SendPhase1$ could be defined to encompass behaviors of sending messages before a minimum amount to make a decision is received by any node. In more complex algorithms message sending would have to be divided into separate operators for separate phases if the τ operator is to be applied as described.

The introduction of a τ operator for TLA$^+$ would mean that specification writers would have to find potential redundant behaviors that could be removed. This would require examining different distributed algorithms and how certain actions can be omitted.

If the τ operator could be applied to any action and also reduced relevant state transitions, the size of the state space could be reduced even more, but the trade-off would be that equivalence would not be ensured. For different distributed algorithms there could exist τ operator application cases in which equivalence is not preserved, but certain safety or liveness properties might be.

1.4.5. Guideline V: restrict the use of data

The fifth guideline defined in [GKO15] comprises three parts that all relate to the use of data in specifications: categorization of data, content queue reduction, and ordering of data buffers.

Data categorization is a form of abstract interpretation that allows structuring a data domain based on categories. All elements in each category represent the same behavior. Instead of working with ranges of values, categories can be used in a specification. This can reduce the state space size of the system model significantly.

For distributed algorithms, the number of received messages is often used to determine whether a node should continue into the next phase of the algorithm or not. In most specifications message sending and receiving are modeled explicitly and all messages are usually added to a set that contains all messages ever sent. This is done because the behavior of a distributed algorithm can be modeled accurately and various properties and execution cases can be evaluated.

However, if there is a need to reduce the state space, the counting of messages could be simplified with data categorization. Instead of counting the cardinality of the message set, it could be divided into categories that represent whether there is a quorum of messages or not.

This would lead to a specification that would only show the general behavior of the algorithm because all nodes would just act based on categories (boolean variables that represent different phases) instead of explicit messages. Nonetheless, such a specification could be useful to determine whether an algorithm has an end state and consensus is always reached.

The next part of the data restriction guidelines deals with reducing the usage of buffers and queues. A traffic light controller system is made τ-confluent by changing information pushing to polling and then applying τ-prioritization and branching bisimulation reduction [GKO15]. The number of states reduced is relatively large and the written modified specification is quite different than the original one.

If it is possible to rewrite a distributed algorithm specification with information polling, then τ-prioritization and branching bisimulation reduction could definitively be applied to that specification. This would allow the creation of a specification with a lot fewer states and behaviors.

The final part of the data usage restriction guideline states that sets should not be used, and instead, ordered buffers are more beneficial for state space reduction [GKO15]. If only a specific order of elements is examined during model checking, then there are fewer possible states and behaviors to explore and verify temporal properties in.

1.4.6. Guideline VI: compositional design and reduction

The sixth guideline defined in [GKO15] describes how the behavior of separate components can be composed and unnecessary actions can be hidden from the specification to reduce the size of the state space. This guideline relies on being able to model the communication of a system in a tree topology. The components that could be given more responsibilities are identified and so the state space is reduced by omitting unwanted behaviors from lower-level components.

Distributed systems can work on various network topologies and the specifications of such systems could also model tree communication topologies. If a distributed algorithm specification is written for the purpose of modeling a specific network topology, this guideline could be considered.

1.4.7. Guideline VII: specify external behavior of sets of sub-components

The last guideline defined in [GKO15] states that the external behavior of a system should be defined first and then the internal behavior can be modeled. This is very similar to the concept of refinement in TLA$^+$ which allows to define abstract specifications and then prove their relations to more lower abstraction level specifications.

In the example provided by the authors, the system model was τ-confluent, however, the τ actions could not be removed with τ-prioritization because they were essential to representing the visible external behavior of the system [GKO15]. This also relates to the point about a specification writer understanding which actions should be τ actions and could be removed. An external behavior that was related to an internal behavior with a conjunction operator caused a significant increase in the size of the state space.

For distributed algorithms, defining more abstract specifications is a preferred method among specification writers. Refinement is then used to relate these abstract specifications with lower abstraction level ones. Despite that, the potential additional state space reduction of τ-confluence in these situations is not recognized enough. If τ-confluence could be applied to refinement specifications, and refinement mappings with omitted τ actions could still be proven to be true, there would be more possibilities to reduce the size of the state space.

1.4.8. TLA$^+$ and mCRL2

When comparing TLA$^+$ and mCRL2, there are a few key differences to consider [GKO15; Lam02; mCR24]. In mCRL2, specifications are constructed by defining actions not as mathematical logical formulas, but as constructs with their data types and parameters. Instead of the propositional logical operators used in TLA$^+$ ($\wedge$ and $\vee$), mCRL2 actions are composed together with the $|$ operator.

An empty action in mCRL2 is written as τ and corresponds to a stuttering step in TLA$^+$ [GKO15; Lam02]. Every τ action is treated as an internal action that is invisible externally, meaning that a mCRL2 multi-action $a \cdot \tau \cdot p$ is equivalent to $a \cdot p$ from a black box perspective [GKO15]. And this corresponds to a more detailed TLA$^+$ specification with a larger state space size being transformed to an abstract specification with fewer states as shown in Figure 6.

In TLA$^+$, when a refinement mapping for a specification is defined, some steps taken in one specification can be mapped to stuttering steps in the other specification. Such circumstances are depicted in Figure 10 where some steps from the $Msgs$ state space correspond to a stuttering step from the $Abst$ state space. Just like τ actions, the steps taken in the $Msgs$ state space could be externally hidden and the only visible steps would be defined in the $Abst$ state space.

1.5. Literature review conclusions

In this literature review, the primary state space reduction methods and state space explosion relief strategies were overviewed while focusing on the distributed system domain. The information and examples provided and discussed in this review are a starting point for potential sources of new TLA$^+$ specification writing guidelines.

LC1. TLAPS and refinement are the most used state space explosion relief strategies among TLA$^+$ specification writers but have their downsides.

LC1.1. Depending on the defined refinement mapping, checking temporal properties only in the abstract specification might not be enough to verify them in the more detailed specification as well. Furthermore, if the state space size of a TLA$^+$ specification is large, other methods would be required to prove refinement mappings effectively.

LC1.2. The task of writing formal mathematical proofs of theorems with TLAPS introduces different problems that must be solved. Formal proofs can also get very large and could potentially have to be rewritten if the TLA$^+$ specification changes considerably.

LC2. In the context of specification writing guidelines, TLAPS gives inspiration for guidelines that could help to write TLA$^+$ specifications whose properties would be proven more easily with TLAPS. This perspective is similar to the “design for verifiability” approach mentioned in [GKO15]: specifications could also be written to be easier to verify with theorem provers such as TLAPS.

LC3. Due to the possible removal of important behaviors from the specification, symmetry sets are not reliable when trying to prove safety properties. They also do not support the verification of liveness properties.

LC4. Running TLC in simulation mode could sometimes be helpful, but not when trying to definitively prove distributed algorithm properties. It could potentially help identify errors in a TLA$^+$ specification faster.

LC5. Partial order reduction could influence how TLA$^+$ specifications are written if Apalache or TLC supported it. Guidelines for writing TLA$^+$ specifications that could greatly benefit from partial order reduction could be formulated for certain distributed algorithms.

LC6. Most specification writing guidelines proposed in [GKO15] coincide with or are similar to popular strategies already used by TLA$^+$ specification writers, while others are not applicable to distributed system TLA$^+$ specifications.

LC7. Writing TLA$^+$ distributed algorithm specifications with information polling instead of information pushing could greatly reduce state space. For most distributed algorithms this would mean that specifications are modified significantly, but certain properties might still be verifiable a lot quicker with the TLC model checker.

LC8. τ-confluence and its similarity to partial order reduction could be applied to TLA$^+$ specifications with the τ operator proposed in Section 1.4.4.1.

2. Reducing state space in TLA$^+$ specifications

This section contains information about TLA$^+$ specifications written based on the conclusions made in the state of the art literature review to achieve the aim of the master's thesis. The full definitions of all the specifications mentioned in this part of the thesis are provided under Appendix 2. Files related to the results of the thesis are provided in a public Git repository as well [Jas24].

Excerpts and various TLA$^+$ formulas used to describe the work done are written based on the specifications in Appendix 2. Additional graphical material is provided along with textual information in this section and in Appendix 1 to better conceptualize and explain the intricacies of the specifications and the algorithms that they model.

The Ben-Or consensus algorithm was chosen as the main distributed algorithm to be examined in the thesis due to its importance in the verification of randomized consensus algorithms [BKL$^+$21; KLS$^+$23]. It is quite a simple algorithm, but it is also interesting to analyze since it uses randomization to determine what values nodes estimate before the next possible round when certain conditions are met.

Randomization introduces a termination property to model checking as TLC does not support probability convergence and would check all unfavorable randomization behaviors explicitly. These circumstances were taken into account when writing specifications and trying to determine how the state and behavior space size of the defined state machine models could be influenced.

Several different specifications of the Ben-Or consensus algorithm were written to try to minimize state space size and gain insights about new TLA$^+$ specification writing guidelines for state space reduction. Ideas conceptualized during the literature review were implemented into specifications in certain aspects, but some problems needed to be solved purely based on the results of additional experimentation and specification modification.

In the following subsections, the Ben-Or consensus algorithm and its characteristics are presented to provide a basis for the rest of the thesis. Then written TLA$^+$ specifications are introduced and the main differences between them are explained to show how they were changed. This is important because changes to specifications give an understanding of what ideas were applied and how these changes can translate to new specification writing guidelines.

Additional information is provided to clarify decisions made related to the properties of the

Ben-Or consensus algorithm and how it was modeled with TLA$^+$. An approach to solving the termination problem caused by estimate value randomization is also described.

2.1. The Ben-Or consensus algorithm

Michael Ben-Or defined an asynchronous randomized consensus algorithm back in the first half of the 1980s [Ben83]. The use of randomization in this algorithm was a proposed solution to the FLP impossibility: there exists no asynchronous consensus protocol that could guarantee binary consensus when at least one process in an asynchronous environment can fail by crashing [FLP85].

Two definitions of the algorithm were provided for scenarios when an adversary is inactive (simple Ben-Or algorithm) and when the adversary tries to do everything to prevent agreement (Byzantine Ben-Or algorithm) [Ben83].

Under the assumption of an active adversary the Ben-Or consensus algorithm guarantees that:

1. No two correct processes decide on different values. This means that consensus is reached among all nodes that are non-adversarial and do not fail.
2. Every process decides on one value and this value is received as input from another process. There can be no decision made on a value val that has not been proposed by at least one other process.
3. Eventually all correct processes will decide on some value. The probability for consensus to be reached over time converges to 1.

The correctness proof of the Ben-Or consensus algorithm has been provided for the case when the number of faulty nodes in an environment is less than half of the number of all nodes ($f < N/2$) and an adversary takes all possible actions to disrupt consensus [AT12]. However, while the algorithm does guarantee agreement and integrity, termination is only theoretically proven. Additional modifications to the algorithm are required to ensure that in a real distributed system the algorithm terminates within a reasonable amount of time and still solves the FLP impossibility.

A typical well-known abstraction in the field of distributed systems for decreasing the execution time of randomized consensus algorithms is the "common coin". In a general sense, its purpose is to provide some value for nodes to base their random estimations on. With a common coin, there are fewer instances of different nodes picking random values that increase the number of rounds required to reach consensus [AT12].

In the case of the Ben-Or consensus algorithm, this can be achieved by making all nodes estimate the same value that was randomly picked by the toss of a global coin [AT12]. Randomization still exists, but it is initiated not by each node separately which leads to less differentiation between randomized node estimate values in the same round.

Algorithm 1 Ben-Or consensus algorithm for process p by Michael Ben-Or [Ben83], adapted from [AT12]

```
Input:
    $f \geq 1;\ f < N/2;\ N \geq 4;$
    $rnd = 0;\ est_p \in \{0,1\};\ rval \in \{0,1\};\ pval \in \{?,0,1\};\ dec_p \notin \{0,1\}$
Output:
    All alive processes decide on the same decision value $dec_p \in \{0,1\}$
 1: while true do
 2:     $rnd = rnd + 1$                                                \\ Current round
 3:     send $(Report, rnd, est_p)$ to all processes                    \\ Phase 1
 4:     wait for $N - f$ $(Report, rnd, rval)$ messages
 5:     if received more than $N/2$ $Report$ messages with the same value $val$ then
 6:         send $(Proposal, rnd, val)$ to all processes
 7:     else
 8:         send $(Proposal, rnd, ?)$ to all processes
 9:     end if
10:     wait for $N - f$ $(Proposal, rnd, pval)$ messages                \\ Phase 2
11:     if received at least $f + 1$ $Proposal$ messages with the same value $val \neq ?$ then
12:         $dec_p = val$
13:     end if
14:     if received at least one $Proposal$ message with the value $val \neq ?$ then
15:         $est_p = val$
16:     else
17:         $est_p =$ random value from $\{0,1\}$
18:     end if
19: end while
```

Algorithm 1 gives a general definition of the Ben-Or consensus algorithm that is applicable in environments where an adversary is passive. Each asynchronous process p executes the statements defined in lines 1–19. Processes are understood as separate nodes in a distributed system that can be in different rounds during the execution of the algorithm. f denotes the number of nodes that can fail by crashing, while N is the total number of nodes.

The est_p value represents a node's initial and current estimate, so it must be set to either a 0 or 1 value before the start of the algorithm in the case of binary consensus. Variables $rval$ and $pval$

describe the possible values of report and proposal messages respectively.

dec_p is the node's decision value and can not be changed once set – if a node makes a decision, it becomes immutable. Since f nodes can fail, the algorithm is executed until all alive processes decide on the same value that is either a 0 or 1.

There are two main phases in each round of the Ben-Or consensus algorithm. During the first phase, all nodes send report messages with their current estimate values to all other nodes. The first phase ends when a node receives enough report messages from other nodes to then send its proposal message. Either the majority of all nodes report the same estimate value and this value is then sent in a proposal message, or a $?$ value is sent as a proposal to indicate no clear candidate decision value.

It is important to mention that a node can send a majority estimate value with a proposal message that is not equal to its current estimate value. Additionally, some nodes can propose $?$ values while others propose estimate values (0 or 1) during the same round. This can occur when a single node reports an estimate value different from all the other node's reported estimate values. The circumstances under which these situations happen are described in Table 1 which contains nodes, and their report and proposal message values.

Table 1. A Ben-Or consensus algorithm situation in which nodes propose different values

Nodes	n1	n2	n3	n4
Report value	0	1	1	1
Proposal value	1	?	?	?

Even though in Table 1 node $n1$ estimated a value of 0 and sent it as a report message to all other nodes, it sends the value 1 in a proposal message because it received three report messages from other nodes with the same value of 1.

Nodes $n2$, $n3$, and $n4$ only received two report messages with the value 1 and one report message with the value 0. Due to there being no value that has been reported by the majority of all nodes, $n2$, $n3$, and $n4$ send the value $?$ in their proposal messages.

One node can not propose a value of 0 if another node proposes a value of 1 in the same round [AT12]. Either a node proposes 0 and any other proposed value is a $?$ or 0, or the node proposes 1 and other proposed values can be $?$ or 1.

During the second phase of the Ben-Or consensus algorithm, all nodes receive at least $N - f$ proposal messages from other nodes and decide on a value or update their current estimate. Nodes can change their decision and estimate values in the same round – the definition provided in Algorithm 1 allows for such an event to occur due to there being two separate if statements in lines 11–13 and 14–18.

There are a total of three distinct possibilities during the second phase for each node:

- The node decides on a new value val and then updates its estimate to the value val. If there are at least $f+1$ proposal messages with the value $val \neq ?$, there also exists a single proposal message that satisfies the condition in line 14.
- The node only updates its estimate value to be equal to val. Because $f + 1 > 1$, a single proposal message with the value $val \neq ?$ will not satisfy the condition in line 11, but the node is required to update its current estimate value to val.
- The node does not change its decision value and updates its current estimate with a random value since there is not a single received proposal message with a valid estimate value of 0 or 1. This means that both conditions in lines 11 and 14 are not satisfied.

Algorithm 4 defines the Byzantine version of the Ben-Or consensus algorithm that guarantees consensus in an environment where the adversary actively tries to malevolently affect the outcome of the algorithm.

It has been proven that the number of adversaries must be less than half of all nodes for agreement to be reached ($f < N/2$) for the simple variant of the Ben-Or algorithm [AT12], which is different than what Michael Ben-Or specified in his paper for the Byzantine version of the algorithm ($f < N/5$) [Ben83]. Since in the [AT12] paper only the simple variant of the algorithm was examined, the Byzantine variant does not necessarily guarantee consensus when $f < N/2$. Hence, Algorithm 4 should guarantee consensus under the assumption that $f < N/5$.

The main differences between the simple and the Byzantine versions of the Ben-Or algorithm all relate to the amount of report and proposal messages that must be received to decide on a value or change the current estimate value of a node.

As described in Algorithm 4, proposal messages with a non $?$ value are sent only if more than $(N + f)/2$ of report messages were received. Decisions can be made when at least $(N + f)/2$ valid proposals have been received, and estimates are not randomized when at least $f + 1$ proposals contain the same value that is 0 or 1. Agreement in the Byzantine version of the algorithm is guaranteed only between nodes that are not adversaries and do not fail by crashing.

A global coin abstraction for the Ben-Or consensus algorithm could be implemented as a synchronized coin that guarantees all nodes that have to estimate random values just change their estimate to the outcome of a coin toss for the current round rnd.

However, if the coin toss is entirely random, its outcome might be unlucky and cause nodes to not reach consensus in fewer rounds than with a lucky value. To make the global coin fortunate, the coin must be tossed not only based on randomness but also on other factors such as the estimations made by other nodes.

An ideal global coin has been shown to guarantee consensus in the simple version of the Ben-Or algorithm if and only if $f < N/3$ [AT12]. The common coin toss value for a round is never randomized in [AT12]: if there is some value val that has been estimated by at least one node, the coin toss is equal to that value val, otherwise, the coin toss value is picked based on the current round number and whether or not a binary consensus value was estimated in the previous rounds.

A less ideal definition of the common coin that has randomization for the coin toss value under certain conditions is also possible and is described as a function in (6).

$$coin(rnd) = \begin{cases} est \in \{0,1\} & \text{if value } est \text{ was estimated by any node in round } rnd \\ tossValue \in \{0,1\} & \text{otherwise} \end{cases} \tag{6}$$

The defined common coin function is supposed to be called in line 17 of Algorithms 1 and 4, and the returned est and $tossValue$ are always the same for a round rnd. This means that any nodes that have to randomly estimate a value in a given round rnd, would instead set their estimate value to est or $tossValue$ which would be the same for all of them in that current round.

2.2. TLA$^+$ specifications

The written TLA$^+$ specifications of the Ben-Or consensus algorithm are based on both simple and Byzantine versions of the algorithm. In several specifications, the less ideal global coin function, as defined in (6), is used to ensure an end state in the specified state machine models and that consensus is reached between correct nodes. In an attempt to reduce the state space size of models for TLC to check, concepts and ideas based on partial order reduction (Section 1.3.1) and the information polling guideline (Section 1.4.1) were used. Other aspects of state formulas such as lenient state conditions (Section 1.1.4) were also taken into account when writing Ben-Or consensus algorithm TLA$^+$ specifications.

Simple Ben-Or algorithm variant specifications were written under the assumption that the number of nodes is $N \geq 4$, and that the number of faulty nodes $f \geq 1$. Byzantine variant specifications assume that the number of nodes is $N \geq 6$, and the number of adversaries is $f \geq 1$.

The number of nodes N and nodes that can fail f can be changed by modifying the configuration files, but each specification also contains $ASSUME$ statements that prevent the violation of input conditions described in Algorithms 1 and 4.

In configuration files for written TLA$^+$ specifications, nodes are given identifiers that are a string of characters in the form of "n*" where * is a single-digit number character. Which nodes are faulty or not can be specified by using these identifiers and adding them to the appropriate set of faulty or correct nodes respectively. All node identifiers must always be added to the correct node set, while the faulty node set has to be a strict subset of the correct node set.

For initial node estimate values, configuration files and the estimate value variables were defined so that TLC would check all possible combinations of initial estimates for all nodes. This can be toggled by changing the value of a defined constant in the configuration file so that the TLC model checker would not have to take into account all possible initial estimates. If turned off, TLC would check only the initial estimates defined in the configuration file and thus would make checking temporal properties for specific initial estimate values more efficient.

The final state of all state machine models defined in Ben-Or TLA$^+$ specifications is considered to be when binary consensus is reached. No two nodes can decide on different values and these decision values must be either 0 or 1. In the Byzantine version of the Ben-Or consensus algorithm, only correct nodes have to reach consensus and the decision values of faulty nodes are ignored when checking the temporal consensus property. TLC is used to check whether all behaviors eventually lead to consensus always being reached among the appropriate nodes.

2.2.1. BenOrMsgs

This TLA$^+$ specification (Figure 23) defines a state machine model of the simple Ben-Or consensus algorithm variant described in Algorithm 1. The specification is written based on general good practices that are widely adopted by distributed algorithm TLA$^+$ specification authors [Git18; Git20].

When TLC is used to check the state and behavior space of a state machine model, it searches for a configuration file with the same name as the specification file by default. The TLC model checker for the `BenOrMsgs` specification should be run with the non-default `BenOr` configuration file depicted in Figure 20.

Lines 2–6 of the configuration file define the main constants and their values during model checking. The $NODES$ set contains all node identifiers, and the $FAULTY\_NODES$ set contains nodes that can fail by crashing.

Initial estimate values are specified by creating two separate sets for nodes that estimate the value 0 ($DECISION\_ZERO$) and nodes that estimate the value 1 ($DECISION\_ONE$).

$CHECK\_ALL\_INITIAL\_VALUES$ can be set to $TRUE$ or $FALSE$. The initial estimate values are used when $CHECK\_ALL\_INITIAL\_VALUES$ is set to $FALSE$, otherwise, TLC will check all possible initial estimates for the current number of defined nodes in the $NODES$ set.

$CHECK\_DEADLOCK$ is used to stop TLC if it encounters a deadlock – a state from which it is impossible to transition to any other state. It is set to $TRUE$ to stop TLC from running indefinitely but does not guarantee the verification of invariants or temporal properties because not all states and behaviors will be checked when TLC is abruptly stopped.

In the `BenOrMsgs` specification all messages sent between nodes are added to the same set

$msgs$ that contains all messages ever sent.

$$\begin{aligned}
&Rounds \triangleq \mathbb{N} \\
&MsgType \triangleq \{\text{“}report\text{”}, \text{“}proposal\text{”}\} \\
&MsgValues \triangleq \{\text{“0”}, \text{“1”}, \text{“?”}\} \\
&MsgRecordSet \triangleq \\
&\quad [Sender : NODES, Round : Rounds, Type : MsgType, Value : MsgValues]
\end{aligned} \tag{7}$$

The definition of a message record that is used in the `BenOrMsgs` specification is provided in (7). Similarly to (4), message records are added to the $msgs$ set, and all nodes can access the $msgs$ variable to count the number of specific messages received. Multiple nodes can send messages of the same type and value since the sender of the message is included in the message record. This can also be done over multiple rounds because round numbers are also sent within messages.

The $msgs$ variable provides a very comprehensive history of node communication when depicted in the state trace generated by TLC after model checking. It can be seen what nodes sent what messages at what round even if node estimate or decision values have not changed over multiple rounds.

Despite that, TLC will not be able to check all possible states of the state machine model because the $Rounds$ formula value does not prevent nodes from sending messages when consensus has not been reached. $Rounds$ is equal to the $\mathbb{N}$ set that is infinite and TLC will have to check an infinite number of states if nodes keep sending messages even though message values can remain the same.

Each state in TLA$^+$ is an assignment of values to defined variables which means that TLC will regard any added elements to the $msgs$ set as new states. Messages with the same values can still be added to the $msgs$ set because they are added in different rounds and are not treated as duplicate elements. If the Ben-Or consensus algorithm will not guarantee consensus due to unfavorable choices in estimate randomization, TLC will keep checking states until a limit of TLC's execution environment resources is reached.

$$\begin{aligned}
&HowManyMessages(sendingNode,\ nodeRound,\ messageType,\ messageValue) \triangleq \\
&\quad Cardinality(\{msg \in msgs : \\
&\qquad \land\ msg.Sender\ \neq\ sendingNode \\
&\qquad \land\ msg.Round = nodeRound \\
&\qquad \land\ msg.Type = messageType \\
&\qquad \land\ msg.Value = messageValue\})
\end{aligned} \tag{8}$$

Nodes can find out the number of messages received from other nodes by counting the number of specific messages at a certain round and excluding themselves as the senders of messages. This is shown in (8) where the $HowManyMessages$ operator is defined for getting the cardinality of a set that contains certain messages that have not been sent by the node $sendingNode$. Similar operators for getting the number of messages of a specific type (proposal or report) are also formulated within the `BenOrMsgs` specification.

$$\begin{aligned}
&\land\, rounds = [node \in NODES\ \mapsto\ 1] \\
&\land\, IF\ CHECK\_ALL\_INITIAL\_VALUES \\
&\quad THEN\ \exists\ initialEstimates \in AllEstimates : estimates = initialEstimates \\
&\quad ELSE\ estimates = [node \in NODES\ \mapsto\ IF\ node \in ESTIMATE\_ZERO \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad THEN\ \text{“0”} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad ELSE\ \text{“1”}] \\
&\land\, decisions = [node \in NODES\ \mapsto\ \text{“}{-1}\text{”}] \\
&\land\, msgs = \varnothing
\end{aligned} \tag{9}$$

The initial state is defined in (9) and is quite typical for distributed algorithm TLA^{+} specifications. In the first state, as per Algorithm 1, the current round is set to 1. All node decision values are set to -1 to indicate that no decisions have been made by any nodes yet. No messages have been sent, so the $msgs$ set is an empty set at the start of the algorithm.

All possible initial estimates are checked when the $IF\ THEN\ ELSE$ statement is equal to the formula with the $\exists$ operator: TLC will check all members of the $AllEstimates$ set that contains all possible initial estimates based on the number of nodes.

If $CHECK\_ALL\_INITIAL\_VALUES$ is set to $FALSE$ in the configuration file, node initial estimates are based on nodes being members of the $ESTIMATE\_ZERO$ set. The state machine model can then be checked with TLC when verifying properties for a single possible set of initial estimate values is satisfactory. In these circumstances it is only required for the specification author to change the value of $CHECK\_ALL\_INITIAL\_VALUES$ and the specification itself does not have to be modified.

$$\begin{aligned} & Next \triangleq \\ & \quad \exists\, node \in NODES : \\ & \qquad \land \neg LastState \\ & \qquad \land \ \lor Phase1(node) \\ & \qquad \quad\ \ \lor Phase2(node) \end{aligned} \tag{10}$$

(10) defines possible actions that can take place in the state machine model after the initial state. Here the $\exists$ operator makes TLC check all possible node actions and all their different orders during model checking. When $CHECK\_DEADLOCK$ is set to $TRUE$ in the configuration file, $LastState$ is used to stop TLC from executing when consensus has been reached by all alive nodes. Due to the $\land$ and $\neg$ operators, the formula defined in (10) will evaluate to $FALSE$ when $LastState$ is $TRUE$ in a state. This will cause a deadlock to occur since no further actions will be taken by nodes.

Operators $Phase1(node)$ and $Phase2(node)$ are defined with the $\lor$ operator and thus could be equal to $TRUE$ in the same step taken by a node. However, additional conditions (Section 1.1.4) prevent $Phase1(node)$ and $Phase2(node)$ from being equal to $TRUE$ at the same time:

- A node can enter the first phase only when it has not sent a report message for its current round and the second phase only when it has sent a proposal message for the current round.
- Nodes do not wait for at least $N/2$ report messages before continuing to the second phase – the number of reports received from other nodes is evaluated after receiving at least $N - f$ report messages and having sent a report message to other nodes (as defined in Algorithm 1).

Additional conditions on node actions allow the use of weak fairness for all node actions since different phase actions can not happen at the same time and can only happen once during the same round. $WF\_vars(Next)$ is enough to guarantee that nodes make progress towards the next phase and round of the algorithm:

- If any node action is enabled, it is enabled continuously until it is performed.
- There is no way for nodes to somehow take back the messages that they have sent thus somehow causing enabled phase actions to become disabled due to the decrease of the received number of messages.
- Only one single node action can be continuously enabled at a time due to strict conditions on action states.

$$\begin{aligned}
&ConsensusReached \triangleq \\
&\quad \land \forall\ node \in NODES : decisions[node] \in \{0, 1\} \\
&\quad \land \forall\ node1, node2 \in NODES : decisions[node1] = decisions[node2]
\end{aligned} \tag{11}$$

$LastState$ defined in (10) is equal to $ConsensusReached$ from (11). Consensus is reached when all node decision values are members of the binary values set (0 or 1) and any two nodes from the $NODES$ set have decided on the same value. Nodes can not change their decision values once they have decided on a 0 or 1 value, so if a node changes its decision value for the first and only time, it is enough to check whether all the decision values of nodes are the same.

2.2.2. BenOrAbstSync

This TLA$^+$ specification (Figure 24) is based on the information polling guideline (Section 1.4.1) and is an attempt to reduce the state space size of the `BenOrMsgs` specification. The main aim of the `BenOrAbstSync` specification is to remove message sending entirely and have nodes look at each other's estimate values directly. Rounds are also omitted from the specification to have TLC show under what circumstances nodes can not reach binary consensus.

If nodes can not be on different rounds, they must all be on the same round that is repeated over and over. During the same round nodes must also wait for a part of other nodes to be in the same phase because there is no separation between a node's current estimate value and the estimate value that would be shared via a report message.

`BenOrAbstSync` is a TLA$^+$ specification of a synchronized Ben-Or consensus algorithm described in Algorithm 2. The synchronized version of the Ben-Or consensus algorithm requires for nodes to wait for each other to complete the first and second phases. This is a must since in this variant of the algorithm nodes never send proposal or report messages to one another.

Algorithm 2 Synchronized Ben-Or consensus algorithm for process p, based on Algorithm 1

```
Input:
    $f \geq 1;\ f < N/2;\ N \geq 4;$
    $proposalValue_p \notin \{?, 0, 1\};\ phaseFlags_p \notin \{1, 2\};$
    $est_p \in \{0, 1\};\ dec_p \notin \{0, 1\}$
Output:
    All alive processes decide on the same decision value $dec_p \in \{0, 1\}$
 1: while true do
 2:     if for more than $N/2$ other processes $p'$: $est_{p'} = val$ then
 3:         $proposalValue_p = val$
 4:     else
 5:         $proposalValue_p = ?$
 6:     end if
 7:     $phaseFlags_p = 1$                                        \\ Process $p$ ends phase 1
 8:     wait until for every other process $p'$: $phaseFlags_{p'} = 1$      \\ Wait before phase 2
 9:     if for at least $f + 1$ other processes $p'$: $proposalValue_{p'} = val$ and $val \neq ?$ then
10:         $dec_p = val$
11:     end if
12:     if for at least at least one other process $p'$: $proposalValue_{p'} = val$ and $val \neq ?$ then
13:         $est_p = val$
14:     else
15:         $est_p =$ random value from $\{0, 1\}$
16:     end if
17:     $phaseFlags_p = 2$                                        \\ Process $p$ ends phase 2
18:     wait until for every other process $p'$: $phaseFlags_{p'} = 2$      \\ Wait before phase 1
19: end while
```

All nodes must wait for all other nodes to finish the first phase to avoid a situation when a node changes its estimate value in lines 13 or 15 before another node evaluates the condition in line 2. There are no rounds and messages to differentiate changed estimate values – nodes must wait for each other to make sure that estimate values are counted at the correct state of the system.

Nodes wait for all other nodes to finish the second phase before starting the first phase again to avoid a case when a node evaluates the condition in line 2 before other nodes have had a chance to change their estimate values in lines 13 or 15.

Algorithm 2 requires the $phaseFlags$ variable to allow nodes to wait for every other node to finish the current phase of the algorithm. $proposalValue$ is necessary due to nodes being able to send proposal messages with a value that is different from their estimate value as defined in

Algorithm 1 and depicted in Table 1.

$$\begin{aligned}
&Phase1(node) \triangleq \\
&\quad \land\ currentPhase = \text{“0”} \\
&\quad \land\ phaseFlags[node] = \text{“0”} \\
&\quad \land\ IF\ CheckPhase1a(node) \\
&\qquad THEN\ SendProposalMessage(node) \\
&\qquad ELSE\ proposalValue' = [proposalValue\ EXCEPT\ ![node] = \text{“?”}] \\
&\quad \land\ phaseFlags' = [phaseFlags\ EXCEPT\ ![node] = \text{“1”}]
\end{aligned} \tag{12}$$

In the `BenOrAbstSync` specification, an additional variable $currentPhase$ is necessary to be able to make sure that nodes wait for each other. (12) shows that instead of nodes waiting for all other nodes to have their phase flags set to 0, the value of the $currentPhase$ variable is used for node synchronization.

Relying on the value of just the $phaseFlags$ variable would be incorrect since some nodes would change their $phaseFlags$ value after they finish the first phase and then prevent the $Phase1$ formula value from being $TRUE$ for other nodes.

$$\begin{aligned}
&Next \triangleq \\
&\quad \lor\ \exists\, node \in NODES : \\
&\qquad \lor\ Phase1(node) \\
&\qquad \lor\ Phase2(node) \\
&\quad \lor\ StartPhase2 \\
&\quad \lor\ NextRound \\
&\quad \lor\ LastState
\end{aligned} \tag{13}$$

$currentPhase$ is part of formulas $StartPhase2$ and $NextRound$ that are used in (13) to change phases of the algorithm. $StartPhase2$ and $NextRound$ are not defined inside the $\exists$ operator block and as such have a *global scope*. These formulas enable additional state transitions to occur that are not actions taken by nodes at a specific step of the state machine model.

Actions during which $StartPhase2$ and $NextRound$ evaluate to $TRUE$ could happen when both $Phase1(node)$ and $Phase2(node)$ are equal to $FALSE$. Here formulas $Phase1(node)$ and $Phase2(node)$ have a *local scope* because they are $TRUE$ when a node performs some operation and changes the state of the system. Both $StartPhase2$ and $NextRound$ are considered to have a *global scope* – they could evaluate to $TRUE$ even when nodes do not act in any way (must wait for other nodes to finish the previous phase) or when nodes do take actions (set proposal values, make decisions, and estimate new values).

The `BenOrAbstSync` specification omits a lot of important behaviors compared to the `BenOrMsgs` specification since nodes can not be in different rounds and must always start the same phase of the algorithm together. Of course, such omissions make the algorithm not guarantee consensus to be reached if at least one node fails by crashing. Nodes would not be able to tell if any other node crashed and are dependent on all nodes to be alive at all times.

$$ConsensusProperty \triangleq \Diamond\Box[ConsensusReached]\_vars \tag{14}$$

Running TLC on the `BenOrAbstSync` specification leads to an error due to the violation of the temporal consensus property defined in (14). $ConsensusReached$ from (11) is used in (14) to make sure that nodes decide on the same value. $ConsensusProperty$ evaluates to $TRUE$ when in an infinite sequence of state transitions eventually there exists such a state from which consensus is always reached.

After a temporal property violation error occurs, TLC returns a state trace that depicts how nodes were able to reach a state from which consensus can never be attained. Based on the information provided by TLC, such a situation was isolated and analyzed, and its circumstances are depicted in Table 2.

Table 2. A Ben-Or consensus algorithm situation in which nodes can never reach consensus

Nodes	n1	n2	n3	n4
Initial estimates	1	0	0	0
Proposal messages	0	?	?	?
New estimates	1	0	0	0

As shown in Table 2, under certain conditions nodes can not possibly decide on the same value in the Ben-Or consensus algorithm. This occurs when node $n1$ decides to randomly estimate a new value that is the same as its previous estimate value. Other nodes in the example do not change their estimate values randomly because they all have received a proposal message from $n1$ with the value 0 that is equal to their estimate values.

In a real distributed system, node $n1$ could at some point in time decide to randomly estimate the value 0. Then nodes would be able to reach consensus and decide on the value of 0. When this would occur at each execution of the Ben-Or consensus algorithm is uncertain.

In TLA$^+$ specifications, all possible behaviors are checked with TLC. It is not possible to omit unfavorable randomization choices and they all have to lead to consensus for the state machine model to satisfy the consensus temporal property defined in (14).

For the consensus property to be valid, the `BenOrAbstSync` specification must be modified to allow all nodes to reach consensus for all possible behaviors of the system. TLC was able to show scenarios that need to be examined when making such modifications and it also gave insight as to why the `BerOrMsgsSet` specification was being model checked for a long time while no real progress was made by nodes to reach an agreement.

2.2.3. BenOrMsgsByz and BenOrAbstSyncByz

Both the `BenOrMsgsByz` (Figure 25) and the `BenOrAbstSyncByz` (Figure 26) specifications are modified versions of the `BenOrMsgs` and `BenOrAbstSync` specifications with the added changes that are defined in Algorithm 4.

Apart from nodes in these specifications having to wait for more messages from other nodes to send proposal messages with binary values, make decisions, or estimate new values, there were also modifications made to the behavior of faulty nodes during the first phase.

In all Byzantine variant Ben-Or algorithm specifications nodes that are faulty always send the $?$ value in their proposal messages. This was done to imitate adversaries that would try to prevent or otherwise affect the result of consensus.

A different configuration file (`BenOrByz` shown in Figure 21) should be used when running TLC on the Byzantine version of the Ben-Or algorithm TLA$^+$ specifications. It is almost identical to the `BenOr` configuration file except the total number of nodes is increased to six, and the initial estimate values are modified slightly.

Byzantine Ben-Or specifications have a Byzantine agreement temporal property that is similar to the definition provided in (11). The one difference is that only non-faulty nodes must reach an agreement, while faulty node decision values are ignored.

Table 3. A situation in the Byzantine Ben-Or algorithm variant when nodes can not reach consensus; $n1$ is a faulty node and always sends $?$ in its proposal messages

Nodes	**n1**	n2	n3	n4	n5	n6
Initial estimates	1	1	1	1	0	0
Proposal messages	?	?	?	?	1	1
New estimates	1	1	1	1	0	0

`BenOrMsgsByz` and `BenOrAbstSyncByz` do not solve the problem of guaranteeing consensus – there still exist behaviors in which nodes can never decide on the same value. Running TLC on the `BenOrAbstSyncByz` specification returned a state trace for the situation depicted in Table 3.

Here node $n1$ is a faulty node and always sends a $?$ value in its proposal message. Nodes $n2$, $n3$, and $n4$ send $?$ values in their proposals since each of them can only discern that there are three estimates with the value 1 and two estimates with the value 0.

The minimum number of estimates with the same value required for sending a proposal message without the $?$ value is at least four: there are six nodes and one faulty node, $(N + f)/2$ would be equal to 3.5, and 4 is the first whole number bigger than 3.5.

$n1$, $n2$, $n3$, and $n4$ then do not randomly change their estimates and instead estimate a new value of 1 because they received at least $f + 1$ proposal messages with the value 1 from nodes $n5$ and $n6$. Only $n5$ and $n6$ randomize their estimate values and it just so happens that they randomly choose to keep their estimate values the same as before.

No nodes make decisions in the situation described in Table 3 since there are only two proposal messages with a value that is not $?$ while there should be at least four of such messages (more than $(N + f)/2$).

2.2.4. BenOrAbstByz

`BenOrAbstByz` (Figure 27) is a modified `BenOrAbstSyncByz` TLA$^+$ specification that allows for nodes to be in different rounds and follows the definition of Algorithm 3.

Algorithm 3 Asynchronous Byzantine Ben-Or consensus algorithm without message sending for process p, based on Algorithm 5

```
Input:
    $f \geq 1;\ f < N/5;\ N \geq 6;$
    $Rounds = \mathbb{N};$
    $\forall\ round \geq 2 \in Rounds : est[round]_p \notin \{0,1\};$
    $\forall\ round \geq 1 \in Rounds : proposalValue[round]_p \notin \{?,0,1\};$
    $rnd = 1;\ est[1]_p \in \{0,1\};\ dec_p \notin \{0,1\}$
Output:
    All non-faulty processes decide on the same decision value $dec_p \in \{0,1\}$
 1: while true do
 2:     wait until for at least $N - f$ other processes $p'$: $est[rnd]_{p'} \in \{0,1\}$
 3:     if for more than $(N+f)/2$ other processes $p'$: $est[rnd]_{p'} = val$ then
 4:         $proposalValue[rnd]_p = val$
 5:     else
 6:         $proposalValue[rnd]_p = ?$
 7:     end if                                        \\ End of phase 1 for process $p$
 8:     wait until for at least $N - f$ other processes $p'$: $proposalValue[rnd]_{p'} \in \{?,0,1\}$
 9:     if for at least $(N+f)/2$ other processes $p'$: $proposalValue[rnd]_{p'} = val$ and $val \neq ?$
    then
10:         $dec_p = val$
11:     end if
12:     if for at least $f+1$ other processes $p'$: $proposalValue[rnd]_{p'} = val$ and $val \neq ?$ then
13:         $est[rnd]_p = val$
14:     else
15:         $est[rnd]_p =$ random value from $\{0,1\}$
16:     end if
17:     $est[rnd+1]_p = est[rnd]_p$
18:     $rnd = rnd + 1$                                 \\ End of phase 2 for process $p$
19: end while
```

In Algorithm 5 nodes can be in different rounds and phases. It is necessary to keep a history of both estimate and proposal values over all rounds due to the following:

- Nodes can not go into the next phase or round if there are less than $N - f$ other nodes in the same phase as them. This is ensured by statements in lines 2 and 8.
- It seems that if the number of nodes $N = 6$ and $f = 1$, estimates and proposals do not need to have their values saved over multiple rounds. Each node in this case would wait for 5 other nodes before continuing, which would essentially prevent any nodes from

being in separate phases or rounds.

- When the number of all nodes $N = 11$ and $f = 2$, nodes can be in different rounds and phases. This is because nodes would wait for only 9 other nodes before proceeding in lines 2 and 8. A single node would be able to stay behind others and be in a different phase or a previous round.
- In the most unfavorable scenario, a single node n_{st1} that stays in the first phase of the first round must have access to all estimate and proposal values across all rounds to eventually catch up to all other nodes. n_{st1} would be able to go into the next phase and round since all other nodes would be ahead of it and the conditions in lines 2 and 8 would be satisfied.

Initial estimate values in the `BenOrAbstByz` specification are stored in a variable that is a function that returns a function that maps node identifiers to their estimate values as defined in (22). To ensure that for the first round node estimate values are set to initial values from the `BenOrByz` configuration file (Figure 21), the $IF\ THEN\ ELSE$ statement had to be modified quite considerably compared to (9) by adding additional nested $IF\ THEN\ ELSE$ statements.

In `BenOrAbstByz`, a finite set of 3 rounds ($\{1, 2, 3\}$) is defined compared to the infinite $\mathbb{N}$ set for round numbers declared in the `BenOrMsgs` specification. With some sort of a common coin abstraction, 3 rounds are enough to reach consensus, and this is verified with TLC in other specifications that have a common coin for randomized estimate values defined in them.

The `BenOrAbstByz` specification still does not guarantee node consensus and differs from the `BenOrAbstSync` specification with the removal of the phase flag variable and the addition of estimate and proposal message value variables that keep values across all rounds. Most importantly, `BenOrAbstByz` does not omit important node behaviors unlike `BenOrAbstSync`, and still does not use a set of records for sending or receiving messages.

2.2.5. Common coin specifications

The `BenOrAbstByzCC` (Figure 28) and `BenOrAbstByzCCL` (Figure 30) TLA$^+$ specifications guarantee consensus with a defined global common coin that is based on the function described in (6).

Both specifications differ in the scope of the common coin toss action: in `BenOrAbstByzCC` the coin is defined as an action with a *global scope* outside the node action formula with the $\exists$ operator in (23) and (24), while `BenOrAbstByzCCL` contains a *local scope* common coin toss action for nodes inside the formula with the $\exists$ operator in (27) and (28).

(23) and (27) define a common coin that ensures that for a particular round the coin value is based on either an already estimated value by a node, or it is randomized and all nodes that have to randomize their estimate values use the value of the common coin. All nodes that the common coin takes into account are non-faulty nodes since in the Byzantine variant of the Ben-Or algorithm only correct nodes must reach consensus.

Conditions on states in both common coin formulas ensure that if a node already estimated some value it did so in the same round as the round for which the coin is about to be tossed. This is checked by requiring the node's estimate value to be set to 0 or 1 for that round. Then it is also ensured that the particular node is actually in the second phase and received at least $f + 1$ proposal values from other nodes.

The possibility of all nodes not estimating a specific value and instead needing to randomize their estimate values is also taken into account in (23) and (27). For this case, it is checked whether all non-faulty nodes have not received at least $f + 1$ proposal values from other nodes. It is also required for the nodes to be in the second phase of the same round as they would have had to wait for $N - f$ proposal values from other nodes (these could all be proposals with $?$ values).

The $\exists$ operator used with the bounded parameter $coinValue$ in (23) and (27) ensures that TLC will check the outcome of all nodes that need to randomize their estimates setting their estimate values to 0, and the other outcome when these nodes set their estimate values to 1.

Strong fairness is not necessary for the common coin toss action and instead, weak fairness is used. If there is a possibility to toss the common coin, it must eventually be tossed, and there is no way for nodes to undo their estimate values to potentially disable the coin toss action.

$$CommonCoinValid \triangleq$$
$$\forall\, round \in NodeRounds : commonCoin[round] \in \{\text{“0”}, \text{“1”}\} \tag{15}$$

$$CommonCoinProperty \triangleq \Diamond\Box[CommonCoinValid]\_vars$$

Eventually, common coins are always tossed for every round during every possible execution of the specification state machine models. This is verified by making TLC check the temporal property $CommonCoinProperty$ defined in (15) while using the `BenOrByzCC` configuration file (Figure 22).

Because all common coin values for all rounds (the specifications model three rounds in total) are set to “-1” in the initial state, the only possibility for the $CommonCoinValid$ formula to evaluate to $TRUE$ is when common coins are tossed in all rounds.

An additional condition in both specifications prevents any node (even faulty nodes) from randomly estimating a value in the $EstimatePhase2c(node)$ operator formula – when at least one node needs to randomly estimate a value, the common coin must be tossed for the algorithm to progress.

The minimum requirement for the common coin to be tossed in a given round is for a single node to not randomize its new estimate value or for all nodes to not have received the required amount of proposals to change their estimates without randomization.

Theoretically, it is possible for the common coin as it is defined to not be tossed during a round: this could occur when at least a single non-faulty node estimated a value and no non-faulty nodes needed to randomize their estimates; a node could just decide to not toss the coin and move to the next round.

However, due to the $CommonCoinProperty$ from (15) being verified by TLC, the common coin is still tossed even in such circumstances where it is optional to toss it. This occurs because the $TossCommonCoin$ action is enabled for more than one state transition and is not disabled and enabled cyclically (as in, enabled then disabled, and then enabled again).

Furthermore, both $TossCommonCoin$ and $SetNextRoundEstimate$ are weakly fair in (25) and (29), and TLC decides to make sure $TossCommonCoin$ is executed before $SetNextRoundEstimate$ since the coin toss action becomes enabled a lot sooner: only one estimate from a node is required for a coin toss to occur compared to needing all nodes to exchange proposal values and make their decisions and estimates before moving to the next round.

Even if the common coin was not tossed during a round when it was not necessary to do so, the overall behavior of the system would still satisfy the temporal Byzantine consensus property.

TLC has verified both $ConsensusProperty$ and the $CommonCoinProperty$ temporal properties for the `BenOrAbstByzCC` and `BenOrAbstByzCCL` specifications. Using weak fairness with the $TossCommonCoin$ action allows for a specification and a state machine model to be more realistic as strong fairness would be harder to ensure in a real-world distributed system.

`BenOrMsgsByzCC` (Figure 34) and `BenOrMsgsByzCCL` (Figure 36) also guarantee consensus, but are based on the `BenOrMsgs` specification that made nodes communicate their estimate and proposal values over report and proposal messages. TLC has verified the $ConsensusProperty$ and $CommonCoinProperty$ temporal properties to be true for these specifications as well.

The `BenOrAbstByzCCLRst` and `BenOrMsgsByzCCLRst` specifications are based on the `BenOrAbstByzCCL` and `BenOrMsgsByzCCL` specifications respectively, and are an attempt to reduce the state space size of Ben-Or state machine models even further.

If in specifications with a common coin toss action of a *global scope* the coin is tossed by no particular node (it is an action taken by the system), then it would also be meaningful to try and put stricter state conditions on the *local scope* common coin actions. Therefore, `BenOrAbstByzCCLRst` and `BenOrMsgsByzCCLRst` only allow a certain non-faulty node to toss the common coin. It does not matter which node performs the common coin toss action – all nodes can make use of the common coin value once it has been set.

In `BenOrAbstByzCCL` and `BenOrMsgsByzCCL`, the common coin can be tossed by any node as defined in (23) and (27).

$$CCNode \triangleq CHOOSE\ node \in NODES : node \notin FAULTY\_NODES \tag{16}$$

Instead of allowing any node to toss the common coin, a certain non-faulty node can be picked by TLC for tossing the coin with the $CHOOOSE$ operator in (16). TLC would pick a single value from the $NODES$ set that is not in the $FAULTY\_NODES$ set and use this same value during model checking (Section 1.1.3). $CCNode$ can then be added as a condition to the common coin toss action as shown in (30).

With these changes, `BenOrAbstByzCCLRst` and `BenOrMsgsByzCCLRst` should describe state machine models with a smaller state space size than `BenOrAbstByzCCL` and `BenOrMsgsByzCCL`. However, there might be situations in which picking a single node for tossing the common coin would not be satisfactory. If a Ben-Or algorithm specification would model the dynamic failure of nodes, then the node that tosses the coin could become faulty, and such a specification could result in a behavior leading to a deadlock.

Nevertheless, TLC has verified the $ConsensusProperty$ and $CommonCoinProperty$ temporal properties for `BenOrAbstByzCCLRst` and `BenOrMsgsByzCCLRst` to be true. Thus, these specifications can be used for the purpose of verifying the properties of the Ben-Or algorithm in the general case when faulty nodes are predetermined and do not change during the execution of the algorithm.

3. Written specification state space reduction

In this section, the written Ben-Or TLA$^+$ specification state space is discussed and examined with examples given on how different specifications increase or decrease the number of states and behaviors in state machine models. Examples are provided to illustrate what TLA$^+$ specification writing decisions impact the state space in meaningful ways. These decisions and other characteristics of different specifications are reflected in the definitions of state space reduction guidelines formulated in Section 4.

Refinement mappings between specifications are also defined in this section. All described refinements have been verified with TLC to ensure that different specifications can be compared to each other in terms of state space size.

The number of behaviors and states of different Ben-Or TLA$^+$ specifications is analyzed to determine what specifications had the most impact on state space reduction of the Ben-Or distributed algorithm state machine models. The insights gained from this analysis are used to make conclusions about specification writing guideline state space reduction effectiveness.

3.1. Sharing values between nodes

All written Ben-Or TLA$^+$ specifications model estimate and proposal value between different nodes in one of two ways: either the values are accessible by nodes directly and without any additional actions, or the values are sent through messages that can be seen by all nodes. The latter approach leads to an increased state space compared to the former way due to the requirement of additional actions for sending messages.

Consequently, `BenOrMsgs*` specifications should have a far larger state space size than `BenOrAbst*` specifications. If such a conclusion would be shown with a state space size comparison of both types of specifications, it would also give more credibility to the information polling guideline proposed in [GKO15] and discussed in Section 1.4.1.

An illustration of the difference between sending messages and sharing node values directly in terms of state space size is provided in Figure 6. This example clearly demonstrates that when estimate values are shared via messages, more actions need to be taken to add these values to the set $msgs$ that can be accessed by all nodes. Each possible order of sending messages is checked by TLC explicitly – the number of behaviors increases quite considerably.

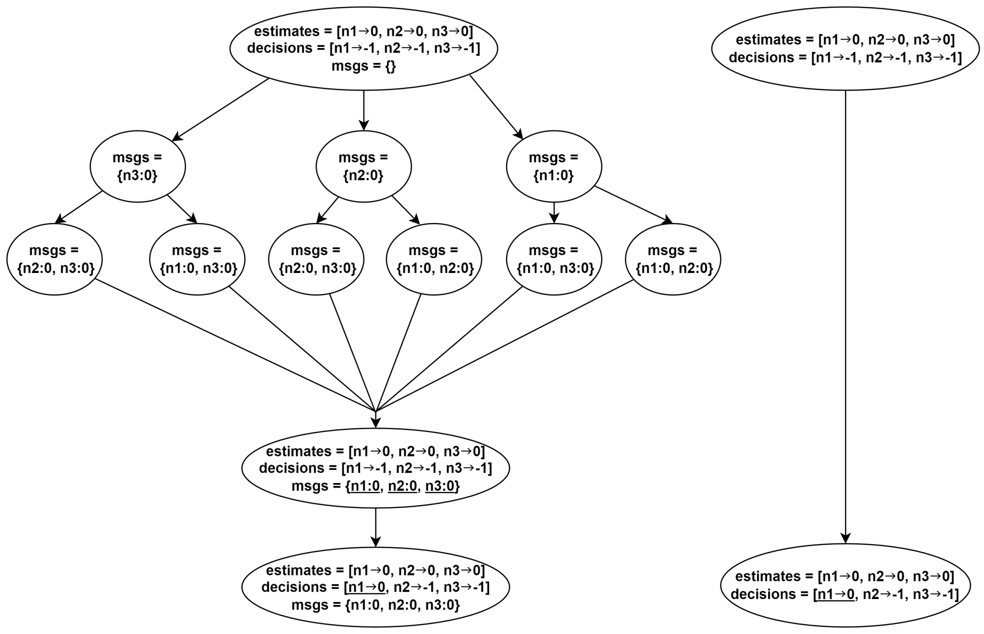
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→-1, n2→-1, n3→-1]
msgs = {}
msgs = {n3:0}
msgs = {n2:0}
msgs = {n1:0}
msgs = {n2:0, n3:0}
msgs = {n1:0, n3:0}
msgs = {n2:0, n3:0}
msgs = {n1:0, n2:0}
msgs = {n1:0, n3:0}
msgs = {n1:0, n2:0}
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→-1, n2→-1, n3→-1]
msgs = {n1:0, n2:0, n3:0}
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→0, n2→-1, n3→-1]
msgs = {n1:0, n2:0, n3:0}
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→-1, n2→-1, n3→-1]
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→0, n2→-1, n3→-1]

Figure 6. State space difference between sending messages (left) and sharing estimate values directly (right)

On the other hand, when estimate values are shared between nodes directly, the number of states and behaviors in the right side of Figure 6 is reduced considerably. Both ways of sharing node estimate values lead to the same final state in which the node $n1$ sets its decision value to 0. The criteria for making a decision for a node are the same in both cases, and the main difference is what TLC would show as a state trace of such an example.

Figure 7 depicts how estimate value sharing through messages provides a more informative state trace of a specific behavior. Compared to the state trace on the right that simply shows that a decision value was changed, the trace of messages that were sent is very useful when examining specifications of distributed algorithms.

Depending on the algorithm, nodes can send messages of different types, and in the case of the Ben-Or distributed consensus algorithm, nodes can send report or proposal type messages. Actually seeing what messages of a certain type were sent with what estimate or proposal values helps to better understand the behaviors that are valid in the state machine model. It also gives additional insight into why certain behaviors lead to a deadlock or a violation of defined temporal properties.

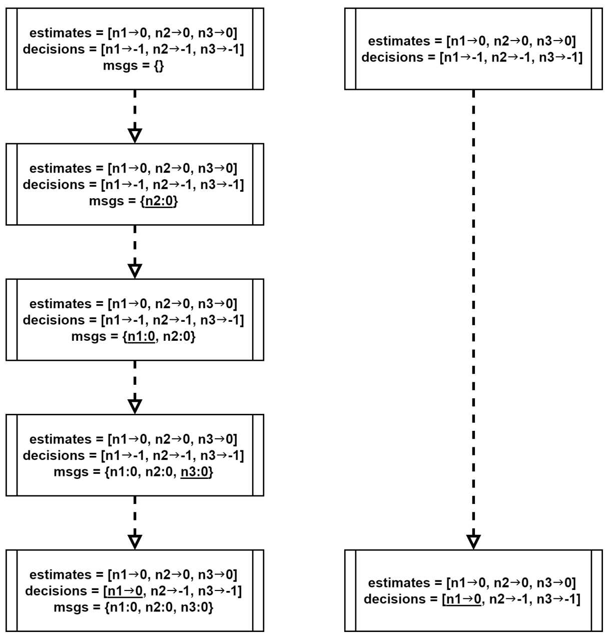
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→-1, n2→-1, n3→-1]
msgs = {}
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→-1, n2→-1, n3→-1]
msgs = {n2:0}
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→-1, n2→-1, n3→-1]
msgs = {n1:0, n2:0}
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→-1, n2→-1, n3→-1]
msgs = {n1:0, n2:0, n3:0}
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→0, n2→-1, n3→-1]
msgs = {n1:0, n2:0, n3:0}
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→-1, n2→-1, n3→-1]
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→0, n2→-1, n3→-1]

Figure 7. State trace difference between sending messages (left) and sharing estimate values directly (right)

3.2. Common coin action scope

`BenOr*CC` and `BenOr*CCL` specifications differ in the state space size of their defined state machine models. In both kinds of specifications, the $TossCommonCoin$ action being of a *local* or *global scope* has an impact on the number of possible states and behaviors.

$$||CCA||NA||CCA||NA||...|| \tag{17}$$

$$||NA||NA||NA||NA||...|| \tag{18}$$

(17) and (18) show the possible sequence of actions if both actions taken by a node (NA) and common coin actions (CCA) are enabled continuously for *global* and *local scope* respectively. When the common coin toss is not a node action in (17) and (24), it can happen before or after an enabled node action NA.

If the common coin toss is a node action in (18) and (28), it can only occur when a node action NA is enabled. There are no additional action sequences to be taken into account since CCA is not a separate action that can be enabled when NA is disabled. When NA is disabled in this situation, the common coin toss action would not be taken. In (17), the common coin action CCA could be enabled while the node action NA is disabled.

Due to the additional conditions that ensure the common coin is always tossed as necessary to guarantee Byzantine consensus, when the common coin toss action has a *global scope* the excess states and behaviors can be omitted by changing its *scope* to *local*.

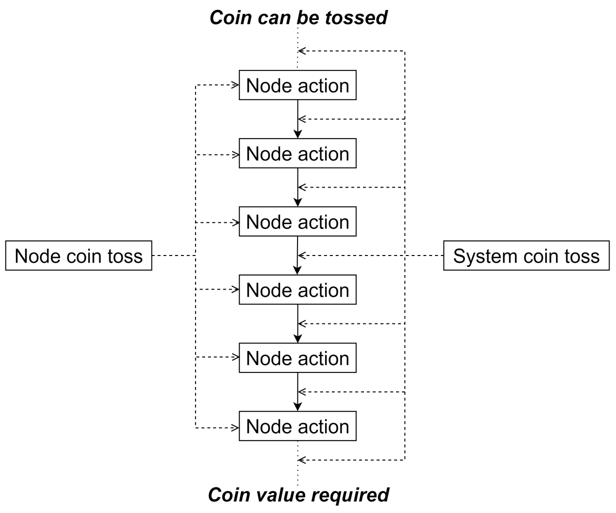

Figure 8. Possible steps in which a *global* or *local scope* common coin toss actions could occur

These changes in possible behaviors are depicted in Figure 8: on the right side the *global scope* system common coin toss action could occur in more steps than the *local scope* node common coin toss action on the left. By making the common coin toss action of *local scope*, unnecessary behaviors are omitted to reduce the size of the behavior and state space.

If the common coin is always tossed and attainable before it is required for the algorithm to progress, as shown in Figure 8, then, ultimately, it does not matter when exactly in a round the common coin is tossed. The additional behaviors that are possible with the system common coin toss actions do not have any impact on the outcome in this situation and can be omitted.

Changes to the state space size due to the *scope* of the coin toss action are a case of partial order reduction (Section 1.3.1) and τ-confluence (Section 1.4.4.1). Irrelevant behaviors are omitted in the `BenOr*CCL` specifications not with a defined τ operator, but by adding additional conditions in the $EstimatePhase2c(node)$ formula and changing the definition of the $Spec$ formula to support a $TossCommonCoin$ action with a *local scope*.

3.3. Tossing the common coin

While the change to the common coin toss action to have a *local scope* reduces the number of possible behaviors (Section 3.2), there are still considerations to be made on how both *global* and *local scope* coin toss actions differ as the number of nodes in a Ben-Or TLA$^+$ specification increases.

When the system tosses the common coin, there is only one possible behavior since the system is both the initiator of the action and that which tosses the coin. Conversely, when nodes have to toss the common coin, there are more possible behaviors since each node is eligible to perform the action.

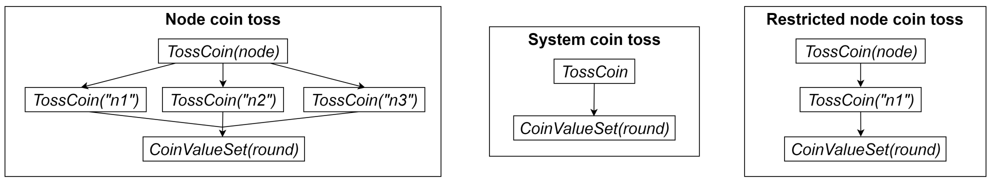

Figure 9. Possible behaviors when tossing the common coin in different actions

This difference in possible behaviors is depicted in Figure 9 with an additional common coin toss action that only allows one specific node to toss the common coin. As the number of nodes in the specification increases, the unrestricted node common coin toss action allows for more behaviors that are unnecessary and increases the state space size.

The increase of nodes also impacts the system common coin toss action, but in a different manner. With more nodes, there are more possible node actions which increase the number of steps in which the *global scope* common coin toss action could occur as shown in Figure 8. However, there is no additional effect on the possible behaviors when performing the common coin toss action itself. From Figure 9, it is evident that the system is the one that tosses the coin and changes in the number of nodes do not impact behaviors in this situation.

By restricting the number of nodes that can toss the coin to 1, and making sure that the node that tosses the coin is non-faulty, the most effective state space size reduction should be achieved in the `BenOr*CCLRst` specifications.

The omission of unnecessary behaviors that do not have an impact on the final outcome here is also a case of partial order reduction (Section 1.3.1) and τ-confluence (Section 1.4.4.1). Restrictions on the *local scope* common coin toss action are made by using the $CHOOSE$ operator to pick a non-faulty node for tossing the coin and adding a stricter state condition in the $TossCommonCoin$ operator defined in (30).

3.4. Specification refinement

If any behavior of a given TLA$^+$ specification A is also a valid behavior in another specification B, then there must exist a refinement mapping that would transform behaviors of A to behaviors of B. When such a mapping is defined and verified, it can be stated that specification A is a refinement of specification B, and all temporal properties that are true for B also hold in specification A.

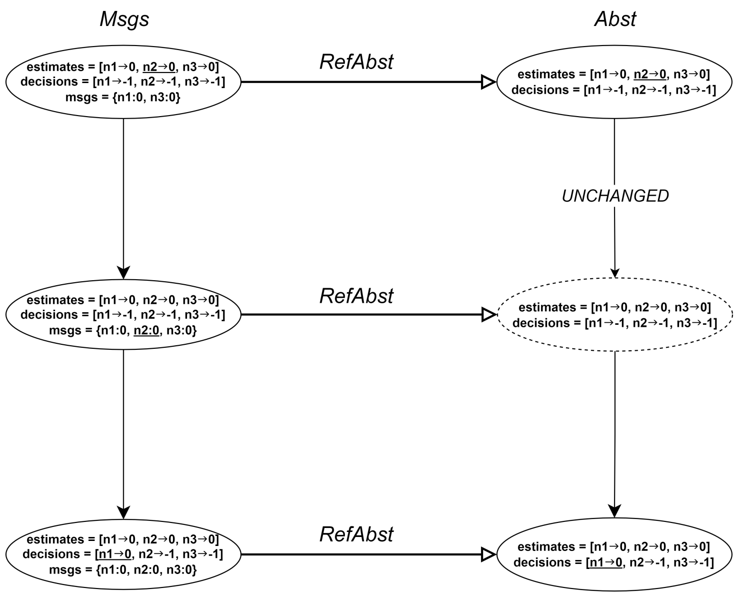
Msgs
Abst
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→-1, n2→-1, n3→-1]
msgs = {n1:0, n3:0}
RefAbst
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→-1, n2→-1, n3→-1]
UNCHANGED
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→-1, n2→-1, n3→-1]
msgs = {n1:0, n2:0, n3:0}
RefAbst
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→-1, n2→-1, n3→-1]
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→0, n2→-1, n3→-1]
msgs = {n1:0, n2:0, n3:0}
RefAbst
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→0, n2→-1, n3→-1]

Figure 10. *Msgs* is a refinement of *Abst* with a defined refinement mapping *RefAbst*

An example of a refinement mapping that would require minimal modification to verify is shown in Figure 10. Here the state and behavior space defined in *Msgs* contains all variables that are present in *Abst*.

The only difference is that *Msgs* models message sending and has an additional $msgs$ variable, but that does not complicate verifying the refinement mapping *RefAbst*. To show that *Msgs* is a refinement of *Abst*, it is enough to map the changes made to the $estimates$ and $decisions$ variables to the corresponding states in *Abst*. A state in which the $msgs$ variable changes can be mapped to a stuttering step in *Abst*, and the refinement mapping *RefAbst* can be verified.

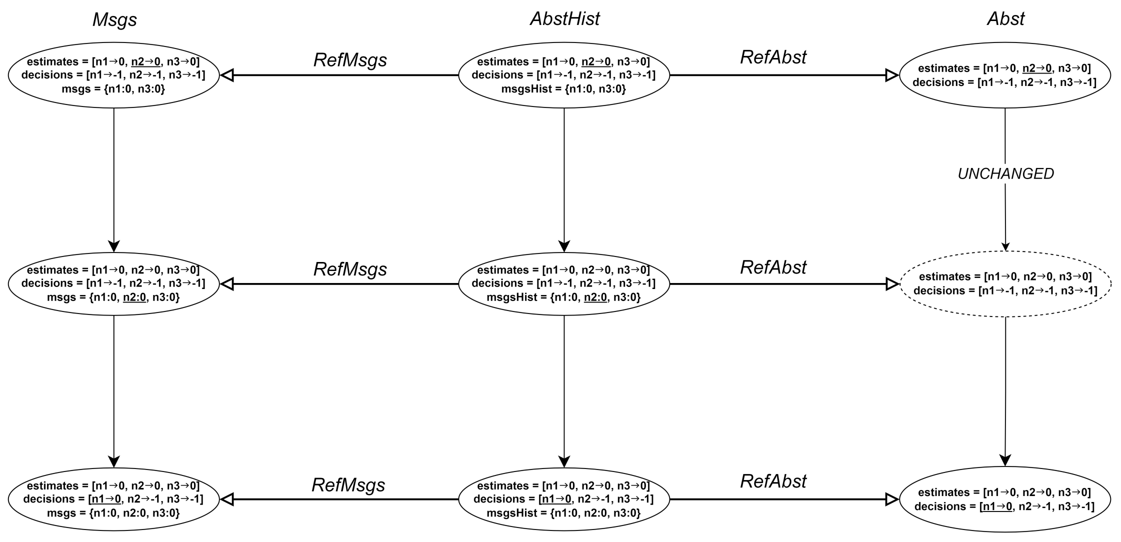
Msgs
AbstHist
Abst
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→-1, n2→-1, n3→-1]
msgs = {n1:0, n3:0}
RefMsgs
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→-1, n2→-1, n3→-1]
msgsHist = {n1:0, n3:0}
RefAbst
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→-1, n2→-1, n3→-1]
UNCHANGED
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→-1, n2→-1, n3→-1]
msgs = {n1:0, n2:0, n3:0}
RefMsgs
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→-1, n2→-1, n3→-1]
msgsHist = {n1:0, n2:0, n3:0}
RefAbst
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→-1, n2→-1, n3→-1]
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→0, n2→-1, n3→-1]
msgs = {n1:0, n2:0, n3:0}
RefMsgs
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→0, n2→-1, n3→-1]
msgsHist = {n1:0, n2:0, n3:0}
RefAbst
estimates = [n1→0, n2→0, n3→0]
decisions = [n1→0, n2→-1, n3→-1]

Figure 11. *AbstHist* is a refinement of *Msgs* and *Abst* with defined refinement mappings *RefMsgs* and *RefAbst*

It is possible to verify a refinement mapping *RefMsgs* for a modified state space of *Abst* as depicted in Figure 11. Here a new state space *AbstHist* with an additional variable $msgsHist$ was introduced to show the refinement mapping *RefMsgs*. This new variable simply adds a new message whenever a node's estimate value changes.

Then *RefAbst* can also be verified to make sure that *AbstHist* still allows behaviors only valid in *Abst*. With both *RefMsgs* and *RefAbst* verified, it can be stated that *AbstHist* is a refinement of *Msgs* and *Abst*.

This means that some valid behaviors in *Abst* could potentially also be valid in *Msgs*, and both of these state spaces share a common subset of valid behaviors defined in *AbstHist*. It would be incorrect to state that *Abst* is a refinement of *Msgs*: certain behaviors valid in *Abst* could be invalid in *Msgs* and vice versa.

If the same temporal properties are true in *Msgs*, *AbstHist*, and *Abst*, the *Msgs* and *Abst* state spaces are comparable with one another. Both of these state space have a shared state space *AbstHist*, but differ from one another in what additional behaviors they enable.

To make conclusions about the state space reduction effectiveness of TLA$^+$ specification writing guidelines, the state and behavior space of specifications that are refinements of one another must be examined. Even if different specifications satisfy the same temporal properties, there is no guarantee that there exists a shared state and behavior space between specifications that also satisfies identical temporal properties.

Without a shared state and behavior space between different specifications, it is unclear whether the differences in their sets of valid behaviors are due to an application of a guideline or just an omission of behaviors that are important and must be valid based on the definition of the Ben-Or distributed consensus algorithm. Unless a refinement mapping is defined and verified, the comparison of such specifications would not necessarily yield valid data relevant to decisions made when writing TLA$^+$ specifications.

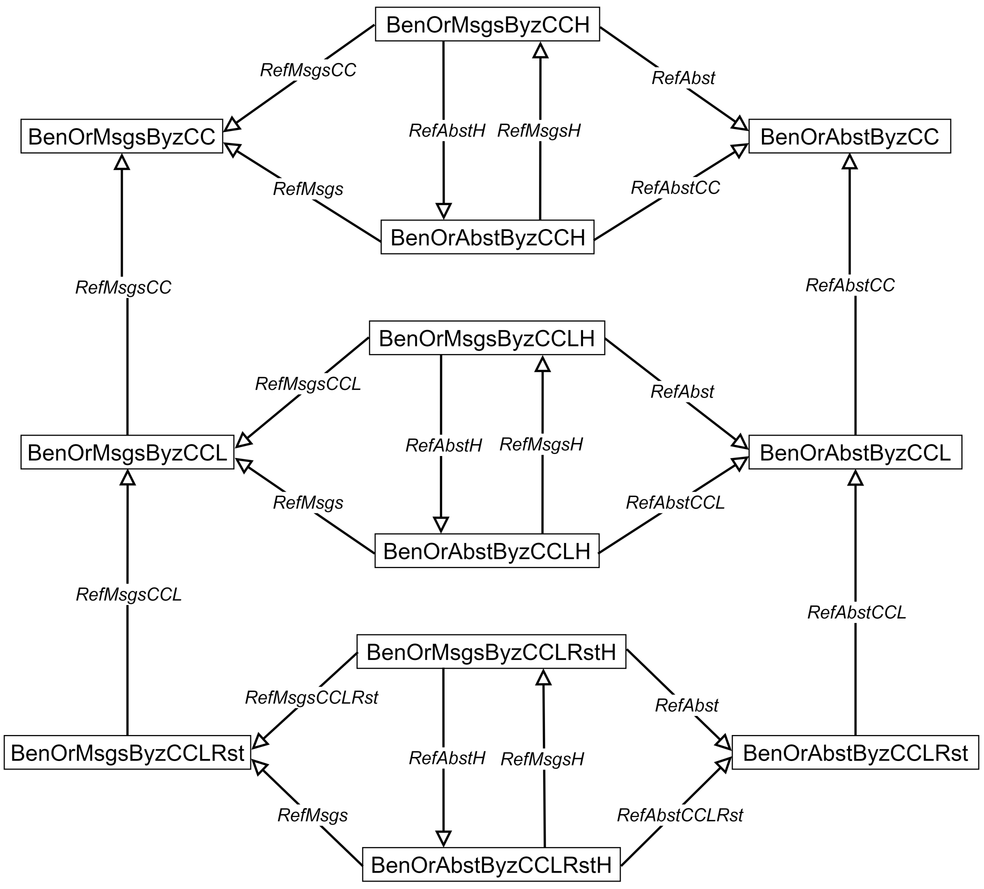

Figure 12. Defined and verified refinement mappings for Ben-Or TLA$^+$ specifications

That is why the refinement mappings depicted in Figure 12 for various written Ben-Or TLA$^+$ specifications have been defined and checked by TLC to be true. This set of defined and verified refinement mappings is sufficient to fully examine state and behavior space size data relevant to TLA$^+$ specification writing guidelines.

Refinement mappings $RefMsgsCC$, $RefMsgsCCL$, $RefAbstCC$, and $RefAbstCCL$ show that the defined local and restricted common coin toss actions reduce the state space and allow behaviors that are a subset of refined specification behaviors. Any changes that were made in the respective specifications did not alter the state space in a way that would invalidate temporal properties or introduce new behaviors that were not present in the refined specifications.

These refinement mappings along with $RefMsgsCCLRst$ and $RefAbstCCLRst$, were verified by defining an $INSTANCE$ statement in (19) and checking whether the behaviors valid in one specification are also valid in the corresponding specification that is being refined.

$$\begin{aligned} &RefMsgsCC \triangleq INSTANCE\ BenOrMsgsByzCC \\ &RefinementProperty \triangleq RefMsgsCC!Spec \end{aligned} \tag{19}$$

TLC checked the $RefinementProperty$ depicted in (19) and defined in appropriate specifications to verify the refinement mapping $RefMsgsCC$. $RefMsgsCCL$, $RefAbstCC$, and $RefAbstCCL$ were also verified similarly and no additional modifications to the respective specifications had to be made (similarly to the situation shown in Figure 10).

To verify refinement mappings between `BenOrMsgsByzCCLRst` and `BenOrAbstByzCCLRst` specifications two new specifications that contain history variables for refinement (Section 1.2.2) were created. These are the `BenOrAbstByzCCLRstH` and `BenOrMsgsByzCCLRstH` specifications (Figures 33 and 39), and they contain additional variables that allowed to verify the refinement mappings $RefAbst$ and $RefMsgs$ with TLC. Other specifications with history variables for refinement shown in Figure 12 were defined in a similar way to `BenOrMsgsByzCCLRstH` and `BenOrAbstByzCCLRstH`.

The refinement mappings $RefAbstCCLRst$ and $RefMsgsCCLRst$ for `BenOrMsgsByzCCLRstH` and `BenOrAbstByzCCLRstH` were checked with TLC to ensure that newly included history variables had not added any unwanted behaviors to the specifications. This is similar to the refinement mappings shown in Figure 11 where a state space with an additional history variable had to be defined.

$$\begin{aligned} &RefAbst \triangleq INSTANCE\ BenOrAbstByzCCLRst \\ &\qquad\qquad WITH\ estimatesAtRound \leftarrow estimateHistory, \\ &\qquad\qquad\qquad proposalsAtRound \leftarrow proposalHistory \\ &RefinementProperty \triangleq RefAbst!Spec \end{aligned} \tag{20}$$

The refinement property checked by TLC for the specification `BenOrMsgsByzCCLRstH` is defined in (20) as $RefinementProperty$. $WITH$ in the $INSTANCE$ statement is used to map the additional history variables added in `BenOrMsgsByzCCLRstH` to the variables of the `BenOrAbstByzCCLRst` specification.

Additional changes had to be made to `BenOrMsgsByzCCLRstH` for the refinement mapping to be verified with TLC. These modifications did not prevent verifying the mapping $RefMsgsCCLRst$.

The changes implemented in `BenOrMsgsByzCCLRstH` to verify $RefAbst$ are the following:

- Addition of the new history variables $estimateHistory$ and $proposalHistory$.
- Whenever a broadcast message is sent, the variables $estimateHistory$ and $proposalHistory$ are updated with the values sent.
- Whenever a node makes a new estimate, $estimateHistory$ is also updated with that value.
- In the initial state, the $estimateHistory$, and $proposalHistory$ variables are set to their initial values as in the initial state of `BenOrAbstByzCCLRst`.
- Defined a new action $SetNextRoundEstimate$ that corresponds to the same action in `BenOrAbstByzCCLRst` and sets the $estimateHistory$ value for the next round of the algorithm.
- Altered the $TossCommonCoin$ action so that it is additionally required for the $estimateHistory$ variable to be set before tossing the coin in the case of a certain node estimating a value without randomization.

In `BenOrAbstByzCCLRstH` the refinement mapping $RefMsgs$ was defined similarly to $RefAbst$ in (20). Additional modifications were made to `BenOrAbstByzCCLRstH` for the refinement mapping to be verified, but these did not invalidate the mapping $RefAbstCCLRst$.

The changes implemented in `BenOrAbstByzCCLRstH` to verify $RefMsgs$ are the following:

- Addition of the new history variables $msgsHistory$ and $estimateHistory$.
- Whenever estimate or proposal values are set, new messages are added to the $msgsHistory$ variable with estimate and proposal values.
- Initial estimate values are sent with report messages and the first phase of the first round can be started after at least $N - f$ of such report messages have been received.
- Whenever a node makes a new estimate, the $estimateHistory$ variable is also changed accordingly.

- In the initial state, the $msgsHistory$, and $estimateHistory$ variables are set to their initial values as in the initial state of `BenOrMsgsByzCCL`.
- Additional actions for checking whether certain messages are present in the $msgsHistory$ variable, and modifications made to ensure that specific messages were sent before allowing a node to enter phase 1, phase 2, or toss the common coin.

Both `BenOrMsgsByzCCLRstH` and `BenOrAbstByzCCLRstH` were shown to be refinements of each other with the refinement mappings $RefMsgsH$ and $RefAbstH$ defined in (31) and (32) checked by TLC.

Based on the verified refinement mappings depicted in Figure 12, all written Ben-Or TLA$^+$ specifications share the same subset of valid behaviors present in the respective specifications with history variables: for `CC` specifications this subset is defined in the `CCH` specifications, for `CCL` specifications the shared subset is defined in `CCLH` specifications, and `CCLRst` specifications share valid behaviors defined in `CCLRstH` specifications.

Furthermore, all valid behaviors of specifications shown in Figure 12 satisfy the same temporal properties of the Ben-Or distributed consensus algorithm: $CommonCoinProperty$ and $ConsensusProperty$ defined in (15) and (26) respectively.

3.5. State space

Table 4. The state space size of state machine models defined by written Ben-Or TLA$^+$ specifications; S – number of all states, and S' – number of distinct states

Nodes	BenOrMsgsByz						BenOrAbstByz					
	CC		CCL		CCLRst		CC		CCL		CCLRst	
	S	S′	S	S′	S	S′	S	S′	S	S′	S	S′
N=6; f=1	5 372 454	1 174 606	3 762 514	552 202	1 374 652	334 910	1 741 222	423 736	793 978	167 232	477 664	120 694
N=7; f=1	42 655 772	8 017 126	33 059 388	3 766 026	10 753 804	2 153 678	7 740 422	1 574 444	3 741 670	637 304	2 227 578	470 896
N=8; f=1	400 710 394	57 229 752	311 618 372	26 415 494	112 084 660	19 092 620	42 582 528	6 641 760	47 416 348	5 695 536	31 010 266	5 553 654
N=9; f=1	2 734 944 408	358 776 292	2 357 787 352	166 795 764	746 174 594	110 739 880	217 926 026	29 694 106	255 851 610	25 858 878	160 511 850	25 297 562

Table 4 shows the state space size of state machine models defined by written Ben-Or TLA$^+$ specifications that have had their Byzantine consensus and common coin temporal properties verified by TLC as described in Table 16.

All specifications in Table 4 are based on the Byzantine variant of the Ben-Or consensus algorithm (defined in Algorithm 4 and modified to have no message sending in Algorithm 3). With common coin action formulas for ensuring agreement between non-faulty nodes, all state machine models were able to reach a state of Byzantine consensus for all possible behaviors in three rounds.

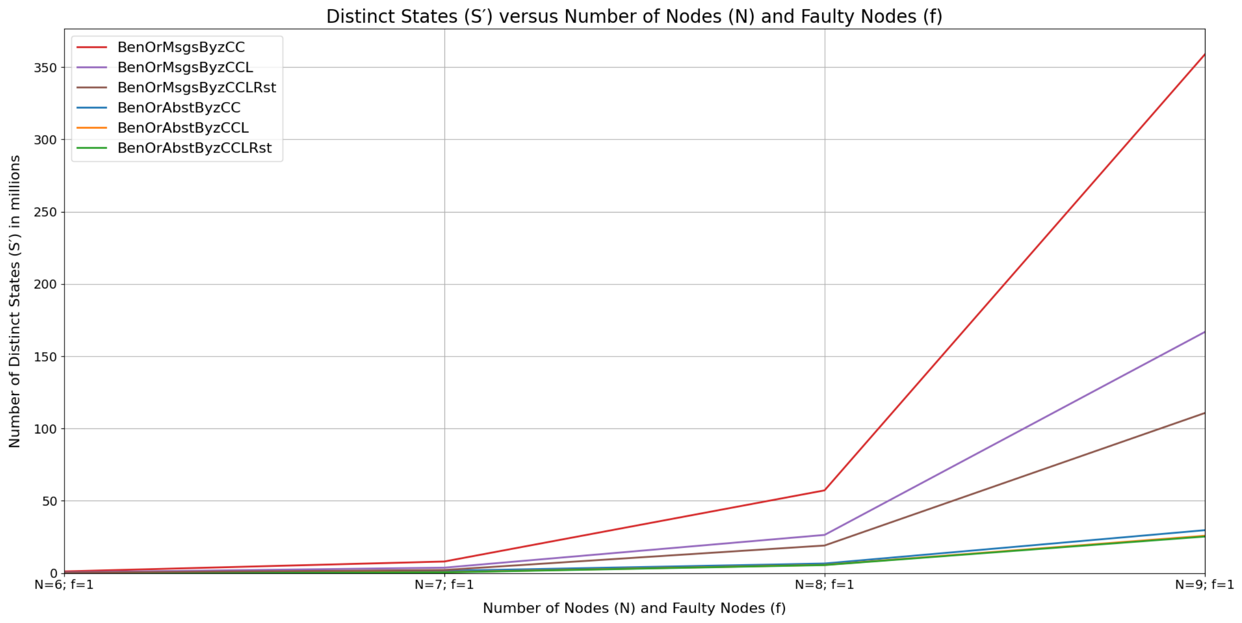

Figure 13. Distinct state space diagram from data in Table 4

Figure 13 shows how the number of distinct states S' of written Ben-Or TLA$^+$ specifications increases as the number of nodes N grows. For both kinds of specifications (`BenOrMsgs` and `BenOrAbst`) the defined local and restricted common coin actions did reduce the amount of distinct states significantly (Figure 8). `BenOrAbst` specifications overall have a smaller distinct state space size due to allowing nodes to evaluate each other's estimate and proposal values without the need to send messages (Figure 6).

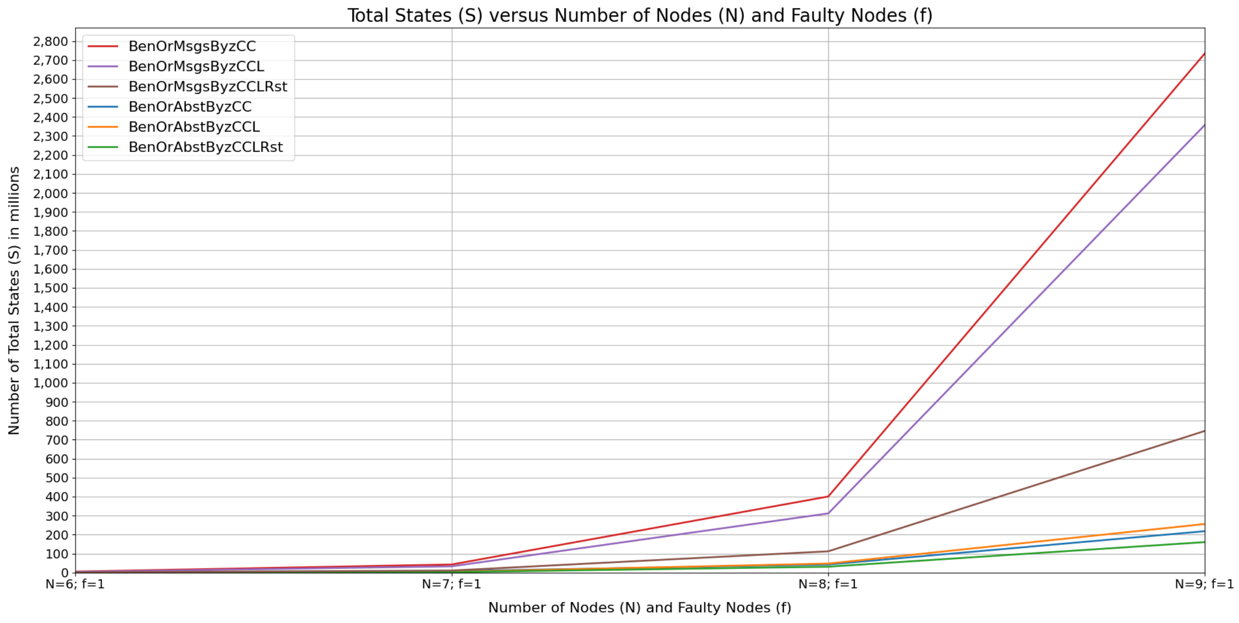

Figure 14. Total state space diagram from data in Table 4

The total number of states depicted in Figure 14 shows different tendencies than in Figure 13. While the `BenOrAbst` specifications still have fewer states and behaviors than `BenOrMsgs` specifications, the `BenOrAbstByzCC` and `BenOrAbstByzCCL` specification space size comparison makes it evident that `BenOrAbstByzCC` has fewer overall states.

This is because, as the number of nodes N increases, the system coin toss action implemented in `BenOrAbstByzCC` does not impact the growth of the state space as much as the local common toss action defined in `BenOrAbstByzCCL` (Figure 9). Figure 19 shows how this occurs when the number of nodes N is increased from 7 to 8.

The higher the number of nodes N, the more there are possible nodes that are eligible to toss the coin in `BenOrAbstByzCCL`. However, in `BenOrAbstByzCC` the increase in the number of nodes only affects opportunities for the coin toss action to occur among node actions. The system tosses the coin in `BenOrAbstByzCC`, and behaviors with coins tossed by different nodes are omitted entirely.

Nevertheless, if the local common coin toss action is restricted to only allow one node to toss the coin (as in `BenOrAbstByzCCLRst`), then the smallest state space size out of all written Ben-Or TLA$^+$ specifications can be attained.

It is also expected that the `BenOrMsgsByzCC` and `BenOrMsgsByzCCL` specification state space size would change similarly: as the number of nodes N would be increased even more (9 and beyond), `BenOrMsgsByzCC` would have fewer states than `BenOrMsgsByzCCL`.

Based on the changes in state space size between different specifications described in Table 4 and in Figures 13 and 14, the following remarks can be made:

- `BenOrAbst` kind specifications have fewer states and behaviors than `BenOrMsgs` specifications in all possbile comparisons. This gives credibility to writing distributed system TLA$^+$ specifications with information polling instead of information pushing. Sending messages by adding them to a set of records (information pushing) leads to a state space of greater size than simply allowing nodes to access each other's estimate and proposal values without sharing them through messages (information polling).
- Making conclusions about TLA$^+$ specification state space size differences requires evaluating both the total and distinct state spaces. The data on the number of distinct states shown in Figures 13 and 18 does not entirely reflect the overall size of the state space for specifications `BenOrAbstByzCC` and `BenOrAbstByzCCL`. With the added states depicted in Figures 14 and 19, `BenOrAbstByzCC` has fewer total states and behaviors than `BenOrAbstByzCCL`, even though `BenOrAbstByzCC` has more distinct states than `BenOrAbstByzCCL`.
- As the number of nodes N increases, the local common coin toss action specifications of the `BenOrAbst` kind grow in the number of states and behaviors more than the global common coin toss action specifications of `BenOrAbst`. When $N > 7$, greater state space reduction is achieved within the `BenOrAbstByzCC` specification rather than within the `BenOrAbstByzCCL` specification.
- The smallest state space size for `BenOrMsgs` and `BenOrAbst` specifications can be achieved by implementing a restricted local common coin toss action that only allows one non-faulty node to toss the coin. `BenOrMsgsByzCCLRst` and `BenOrAbstByzCCLRst` has the smallest number of behaviors and states compared to the specifications of the same kind. These specifications also do not increase in state space size as much as `BenOrMsgsByzCCL` or `BenOrAbstByzCCL`.

4. State space reduction guidelines

This section contains state space reduction guideline definitions based on the decisions made when writing Ben-Or TLA$^+$ specifications and how that changed the state space size (Section 3).

All guidelines have been defined in the same form and each has seven distinct parts that cover different aspects of guideline recognition and application in TLA$^+$ specifications.

These seven distinct parts are grouped into the "essentials" and "additions" categories for greater guideline definition clarity and better guideline comparability. Essentials contain the parts that are crucial for guideline understanding and differentiation between other guidelines. Additions cover the parts of guidelines that provide supplementary information that is intended to be of most use to TLA$^+$ specification authors.

The essentials of each guideline are the following parts:

- *Description* – the general definition of a guideline in textual form.
- *Required definitions* – TLA$^+$ statements that must be present in the specification to apply the guideline.
- *Application properties* – safety and liveness properties that must be valid after guideline application in a TLA$^+$ specification.

Additions of each guideline encompass these parts:

- *Important aspects* – practical concerns that arise when applying the guideline in a TLA$^+$ specification.
- *Advantages* – potential positive effects on the TLA$^+$ specification after guideline application.
- *Drawbacks* – potential negative effects on the TLA$^+$ specification after guideline application.
- *Application cases* – Ben-Or TLA$^+$ specifications that satisfy the Byzantine consensus property, have their state space size data shown in Table 4, and in which the corresponding guideline was applied.

Furthermore, the state space reduction effectiveness of guidelines is calculated and provided in this section based on the data presented in Table 4. Defined and verified refinement mappings described in Section 3.4 are used to correctly identify which specification state space sizes can be compared with one another and what guideline application they represent.

A description of an algorithm to detect the applicability and correctness of defined state space reduction guideline application is also provided in this section. Additionally, an implementation that is able to detect the applicability of the node actions guideline in TLA$^+$ specifications is presented as well.

4.1. Guideline definitions

4.1.1. Information pushing

Table 5. Information pushing guideline essentials

Information pushing essentials		
Description	**Required definitions**	**Application properties**
All nodes should communicate via messages that are added to a shared set of all sent messages as records.	`VARIABLE msgs` `RecordSet ≜ [field: value, ...]` `Send(message) ≜` `∧ message ∈ RecordSet` `∧ msgs' = msgs ∪ message`	`□(msgs ⊆ RecordSet)` `□(Send(message) ⇒ message ∈ msgs')`

Table 6. Information pushing guideline additions

Information pushing additions			
Important aspects	**Advantages**	**Drawbacks**	**Application cases**
Messages must have round numbers and sender identifiers to allow to send messages with the same content multiple times over different rounds.	TLC state trace contains all communication between nodes in an easy to understand form.	Harder to use TLAPS with record type variables.	BenOrMsgsByzCC, BenOrMsgsByzCCL, BenOrMsgsByzCCLRst.

Tables 5 and 6 show the information pushing guideline and its main parts. This way of writing distributed algorithm TLA$^+$ specifications is common as most well-known and widely used distributed algorithm specifications have this guideline applied to them [Git18; Git20].

It is expected for this guideline to be applied when its advantages are more beneficial to the TLA$^+$ specification author than its increase to the number of states and behaviors. Figure 6 illustrates how having message variables adds a lot of states that could be omitted by applying the information polling guideline instead. Nevertheless, the TLC state trace with the information pushing guideline applied provides more insight into what behaviors are valid in a TLA $^+$ specification than in the case of the information polling guideline (Figure 7).

There are a couple more noteworthy intricacies of the information pushing guideline that would help in applying it more optimally.

If the amount of fields used in the record for message communication is minimized, the state and behavior space size growth when applying the information pushing guideline decreases accordingly.

While the sender and round number field importance is mentioned in *important aspects*, the usage of such fields in a record heavily depends on what kind of distributed algorithm a TLA$^+$ specification models.

Round numbers are almost always useful, but sender identifiers could be omitted and a count field could be used to count the number of messages received. In such a case, the cardinality of the set of all sent messages would be the number of all distinct messages ever sent.

Records are indeed harder to work with when trying to construct mathematical proofs in TLAPS using the Zenon or Isabelle backends [Inr24b]. However, using a different backend could help to mitigate the effects of this drawback.

4.1.2. Information polling

Table 7. Information polling guideline essentials

Information polling essentials		
Description	**Required definitions**	**Application properties**

All nodes should share their internal variable values with one another directly without the need to send them via messages.	`CONSTANT NODES`
	`VARIABLE nodeValues`
	`ShareValue(node) ≜`
	    `nodeValues' = [nodeValues EXCEPT ![node] = "1"]`
	`HowManySharedValues ≜`
	    `Cardinality({node ∈ NODES: nodeValues[node] ≠ "-1"})`
	`Init ≜`
	    `nodeValues = [node ∈ NODES ↦ "-1"]`
		`NODES ≠ ∅`
		`◊□(∀ node ∈ NODES: nodeValues[node] ≠ "-1")`

Table 8. Information polling guideline additions

Information polling additions			
Important aspects	**Advantages**	**Drawbacks**	**Application cases**
Nodes must discern when certain variable values are accessible. This can be achieved by using negative values ("-1") to denote that a certain value had not been shared and is not to be accessed yet.	Records do not have to be used for the purpose of sharing node internal variable values.	It is required to keep the history of node internal variable values across rounds and phases so that nodes that fall behind can catch up with others.	BenOrAbstByzCC, BenOrAbstByzCCL, BenOrAbstByzCCLRst.

The information polling parts shown in Tables 7 and 8 define a guideline which when applied would potentially reduce the state space size of specifications more significantly than compared to information pushing (Figure 6).

This guideline relies on using some sort of values for node internal variables to let other nodes understand when certain values can be accessed. If node estimate and decision values are always available to be examined by other nodes, somehow nodes must not look up variable values that would allow them to not abide by the Byzantine Ben-Or consensus algorithm definitions (Algorithms 4 and 5).

In all applications cases of the information polling guideline, the value "-1" is used to denote that a particular node has not entered a phase or round yet. The estimate and decision values are stored within variables that keep a history of all node internal variable values throughout all possible rounds.

This means that if the number of rounds is not finite, the application of the information polling guideline becomes more difficult to properly realize. However, the omission of record type variables and message sending from TLA$^+$ specifications causes a significant reduction of state space, which means that the time for TLC to check temporal properties also reduces considerably.

All these intricacies of the information polling guideline have to be taken into account before trying to apply the guideline to a TLA$^+$ specification in an optimal way. Nonetheless, the smaller state space size can be well worth the effort put in to implement information polling.

4.1.3. System actions

Table 9. System actions guideline essentials

System actions essentials		
Description	**Required definitions**	**Application properties**

Actions should be defined so that they are taken by the system rather than being taken by nodes.	`VARIABLE var`
	`SystemAction ≜ ...`
	`Next ≜`
	`    SystemAction`
		`□(SystemAction ⇒ var' ≠ var)`

Table 10. System actions guideline additions

System actions additions			
Important aspects	**Advantages**	**Drawbacks**	**Application cases**
The fairness of a node action must stay the same when it is transformed into an action taken by the system.	No dependency on a specific node to take a system action as the system can always change the values of variables on its own.	Additional variables or conditions on the state might be required to control when system actions can be taken.	BenOrMsgsByzCC, BenOrAbstByzCC.

The system actions guideline (defined in Tables 9 and 10) is promising in terms of its greatest advantage – the independence from nodes in system actions and how that translates to an omission of unnecessary behaviors (Figure 9).

It was possible to apply the system actions guideline in written Ben-Or TLA$^+$ specifications due to the decision to define a common coin toss action. Furthermore, this action was formulated to occur for rounds, not specific nodes.

If a defined system action would have a direct dependence on nodes, then the application of this guideline would not be optimal.

A system action could allow for a particular node to change its value by having an $\exists$ or $\forall$ bounded operator in its definition that would relate the action to a set of node identifiers. In that case, the system action would have a node action inside of it, which would not be meaningful in terms of state space size reduction.

To avoid this, it is required to have something else other than node identifiers for state conditions inside system actions.

In the written Ben-Or TLA$^+$ specifications, the global common coin toss action (23) relates itself to rounds with a bounded $\exists$ operator. This makes sense as the common coin has to be tossed for a round and is available to all nodes once tossed.

Not all distributed algorithm TLA$^+$ specifications require a common coin, but there can still be some variables that could be changed by the system in a meaningful way that does not relate to specific nodes.

4.1.4. Node actions

Table 11. Node actions guideline essentials

Node actions essentials		
Description	**Required definitions**	**Application properties**

Actions should be defined so that they are taken by nodes rather than being independently taken by the system.	`CONSTANT NODES`
	`Action(node) ≜ ...`
	`Next ≜`
	`    ∃ node ∈ NODES:`
	`        Action(node)`
		`NODES ≠ ∅`
		`□(node ∉ NODES ⇒ ¬Action(node))`

Table 12. Node actions guideline additions

Node actions additions			
Important aspects	**Advantages**	**Drawbacks**	**Application cases**
A system action's fairness must stay the same when it is transformed into an action taken by nodes.	Possible to use the identifier of the node that performs the action inside the action formula.	Might need to modify the specification with additional state conditions on other action formulas.	BenOrMsgsByzCCL, BenOrMsgsByzCCLRst, BenOrAbstByzCCL, BenOrAbstByzCCLRst.

Tables 11 and 12 define all seven main parts of the node actions guideline. Most actions in distributed algorithm TLA$^+$ specifications are defined as being taken by nodes [Git18; Git20] – this is a natural way of modeling real-world distributed systems with TLA$^+$ since nodes are separate from one another and take actions individually.

In the *description* part of the guideline, actions that are "independently taken by the system" refer to actions that are not related to specific nodes (like the global common coin toss action shown in Figures 8 and 9).

Node actions guideline drawbacks are related to additional state conditions that had to be added in order for the local common coin toss action to not invalidate the Byzantine consensus property in written Ben-Or TLA$^+$ specifications.

Additional state conditions might also be required for the application of the system actions guideline, but which specification writing guideline out of the two poses fewer disadvantages depends on the particular TLA$^+$ specification and the intent of its author.

It might be that adding a node action in most cases is simpler and requires fewer changes to be made to the specification, than adding a system action.

Nevertheless, the opposite could also be true, and not all ways of applying the node actions guideline could be more optimal in state space reduction than some applications of the system actions guideline (Figure 9).

4.2. Comparison to guidelines proposed in [GKO15]

A few out of seven guidelines proposed for mCRL2 specifications [GKO15] are similar to the guidelines defined in Section 4.1.

Information polling [GKO15] directly corresponds to the information polling guideline: the sensors described in [GKO15] had to keep information about being triggered which is similar to nodes setting their estimate and proposal values from "-1" to a binary consensus value.

Using global synchronous communication [GKO15] is related to the information pushing guideline. If all messages are added to a shared set $msgs$, then the situation described for global synchronous communication is avoided as nodes do not have to communicate with each other through specific intermediary nodes.

The confluence and determinancy guideline [GKO15] is similar to the system actions and node actions guidelines. Removing unneeded message sending behaviors with partial order reduction (Figures 3–5) is akin to omitting unnnecessary commoin coin toss behaviors (Figure 8). In both cases steps that do not change the overall behavior of the system are treated as τ steps and can be omitted to reduce the state space size.

4.3. Guideline state space reduction

To show guideline state space reduction effectiveness, the state space sizes of appropriate Ben-Or TLA$^+$ specifications must be compared.

Table 13. Relational structure of written Ben-Or TLA$^+$ specification domain objects

TLA$^+$ specification	Valid Ben-Or properties	Shared state space	Primary guideline	Secondary guideline	Common coin scope
BenOrMsgsByzCC	CommonCoinProperty, ConsensusProperty	BenOrMsgsByzCCH, BenOrAbstByzCCH	Information pushing	System actions	Global
BenOrMsgsByzCCL	CommonCoinProperty, ConsensusProperty	BenOrMsgsByzCCLH, BenOrAbstByzCCLH	Information pushing	Node actions	Local
BenOrMsgsByzCCLRst	CommonCoinProperty, ConsensusProperty	BenOrMsgsByzCCLRstH, BenOrAbstByzCCLRstH	Information pushing	Node actions	Local restricted
BenOrAbstByzCC	CommonCoinProperty, ConsensusProperty	BenOrMsgsByzCCH, BenOrAbstByzCCH	Information polling	System actions	Global
BenOrAbstByzCCL	CommonCoinProperty, ConsensusProperty	BenOrMsgsByzCCLH, BenOrAbstByzCCLH	Information polling	Node actions	Local
BenOrAbstByzCCLRst	CommonCoinProperty, ConsensusProperty	BenOrMsgsByzCCLRstH, BenOrAbstByzCCLRstH	Information polling	Node actions	Local restricted

Relevant Ben-Or TLA$^+$ specification comparability is derived from the relational structure depicted in Table 13. Here a subset of all written Ben-Or TLA$^+$ specifications are treated as objects from the same domain (written Ben-Or TLA$^+$ specifications) that are to be compared in terms of their similarities and differences [GS12].

The properties of objects in Table 13 correspond to the refinement mappings defined and verified with TLC in Figure 12, and the application cases for defined guidelines described in Tables 6, 8, 10 and 12.

To select distinct Ben-Or TLA$^+$ specification pairs for the evaluation of information polling and node actions guideline application state space reduction effectiveness, the following comparison criteria were applied for each possible object pair shown in Table 13:

CC1. The valid Ben-Or properties of both objects must be the same.

CC2. If the primary guideline of both objects is the same, then the shared state space, secondary guideline and common coin scope must differ between both objects.

CC3. If the primary guideline of both objects is different, then the shared state space, secondary guideline and common coin scope must be the same between both objects.

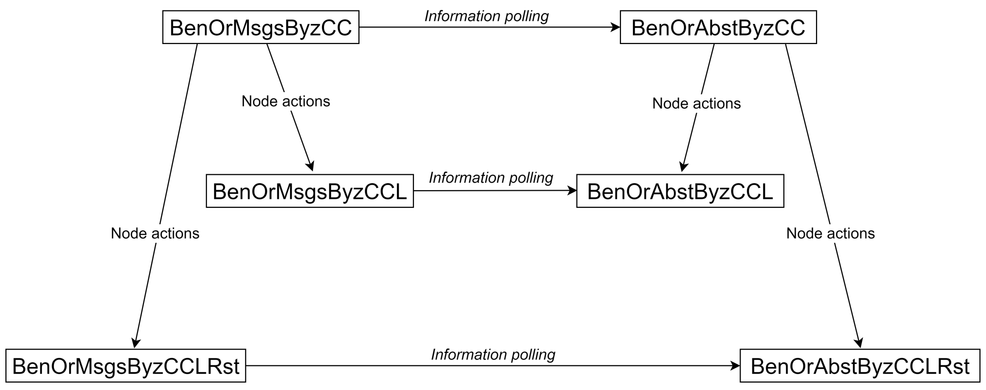

Figure 15. Specification state space relations in terms of applied guidelines

After applying the criteria **CC1**, **CC2**, and **CC3** to objects from Table 13, the pairs of Ben-Or TLA$^+$ specifications to be compared shown in Figure 15 were attained. For pairs depicted in Figure 15, the total state space change will be evaluated to show how guideline application decreased the number of states.

$$SR = \frac{S_e}{S_r} \tag{21}$$

To ascertain the state space reduction effectiveness of information polling and node actions guidelines, the state space reduction property SR will be used and calculated based on its definition in (21).

S_e is the total number of states in a TLA$^+$ specification e that has an “expanded” amount of states and behaviors, some of which could be omitted. S_r is the total number of states in a TLA$^+$ specification r that has a reduced state space size due to an application of a state space reduction guideline.

SR indicates how many times the state space size was decreased by an application of a guideline. The number of total states is used in the definition of SR since distinct states do not fully represent the state and behavior space of state machine models (discussed in Section 3.5).

Table 14. Information polling guideline state space reduction

Nodes		**Information polling guideline *SR***		
	e	BenOrMsgsByzCC	BenOrMsgsByzCCL	BenOrMsgsByzCCLRst
	r	BenOrAbstByzCC	BenOrAbstByzCCL	BenOrAbstByzCCLRst
N=6; f=1		3.085	4.739	2.878
N=7; f=1		5.511	8.835	4.828
N=8; f=1		9.410	6.572	3.614
N=9; f=1		12.550	9.215	4.649

The state space reduction property SR data described in Table 14 shows that overall the information polling guideline application reduced the state space size of comparable Ben-Or TLA$^+$ specifications quite considerably. These calculations should be evaluated taking into account not just how many times the state space was reduced, but also the huge amount of states and behaviors that was expressed in Table 4.

12.550 times reduction of states when comparing `BenOrMsgsByzCC` and `BenOrAbstByzCC` specifications is a significant achievement for the information polling guideline. Other SR values are impressive as well, and for all possible pairs of specifications e and r there were no values lower than 1.0 (this would indicate a state space size increase instead of a reduction).

There is a considerable decrease in information polling guideline state space reduction effectiveness when $N = 8$. `CCL` and `CCLRst` specifications both experience a decrease in state space reduction, but the SR value increases again when $N = 9$.

It is unclear why this happens when $N = 8$ and why it is not represented in the `CC` specification pair. There is a possibility that the addition of local common coin toss actions to specifications somehow correlates to even numbers of nodes.

Another consideration would be that for a certain amount of nodes, there are not enough additional behaviors that can be omitted, and as such the state space reduction property SR decreases under these conditions more significantly.

Table 15. Node actions guideline state space reduction

Nodes		Node actions guideline ***SR***			
	e	BenOrMsgsByzCC	BenOrMsgsByzCC	BenOrAbstByzCC	BenOrAbstByzCC
	r	BenOrMsgsByzCCL	BenOrMsgsByzCCLRst	BenOrAbstByzCCL	BenOrAbstByzCCLRst
N=6; f=1		1.428	3.908	2.193	3.645
N=7; f=1		1.290	3.967	2.069	3.475
N=8; f=1		1.286	3.575	**0.898**	1.373
N=9; f=1		1.160	3.665	**0.852**	1.358

The node actions guideline application state space reduction effectiveness depicted in Table 15 showcases how applying the same guideline in different ways can improve the SR property.

When comparing `CC` and `CCL` specification pair SR values, it is expected that as the number of nodes N increases, the node actions guideline will be less effective than the system actions guideline (Figure 9).

`BenOrAbstByzCC` and `BenOrAbstByzCCL` specification SR values when $N \geq 8$ are lower than 1.0, meaning that applying the node actions guideline without restricting the common coin toss action is not optimal in terms of state space size reduction.

The comparison of `CC` and `CCLRst` specification pairs shows that defining a restricted local common coin toss action is more effective when trying to reduce the state space as much as possible. This approach of applying the node actions guideline does have its own drawbacks (discussed in Section 2.2.5), but `CCLRst` specifications are suitable for general Byzantine consensus property verification.

4.4. Guideline detection

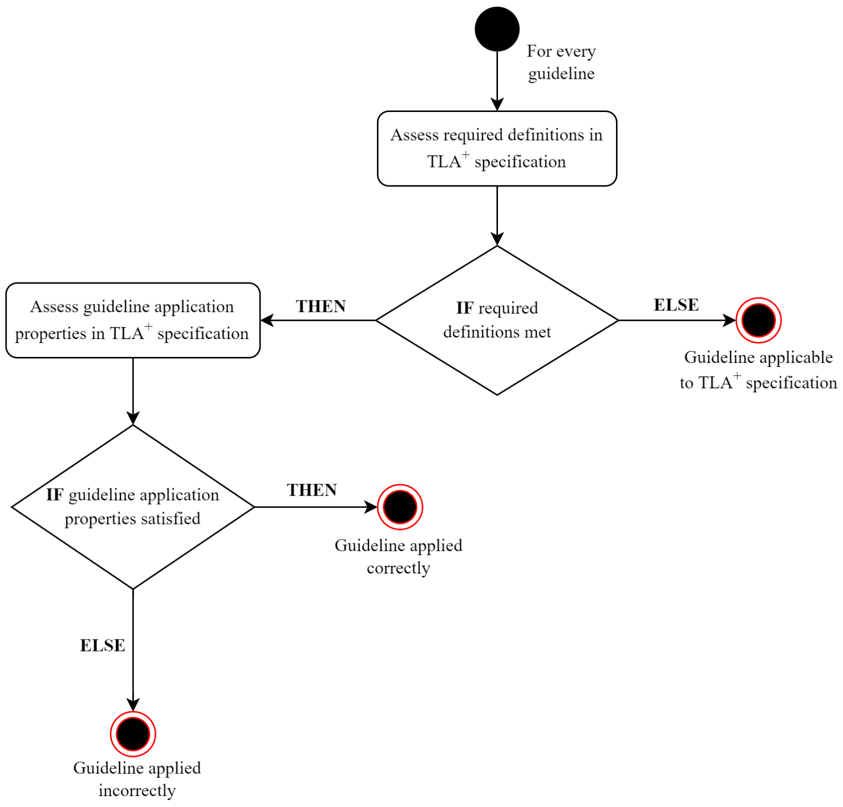
For every guideline
Assess required definitions in TLA+ specification
IF required definitions met
THEN
ELSE
Guideline applicable to TLA+ specification
Assess guideline application properties in TLA+ specification
IF guideline application properties satisfied
THEN
Guideline applied correctly
ELSE
Guideline applied incorrectly

Figure 16. Flowchart of an algorithm to detect guideline applicability and whether guidelines were applied correctly

The algorithm depicted in Figure 16 would detect guidelines and assess their application correctness based on the TLA$^+$ statements and formulas provided in the *required definitions* and *guideline application properties* parts of specification writing guidelines.

Since both defined guidelines can be applied in the same specification, the algorithm would be able to recognize that multiple guidelines are present as the assessment of definitions and application properties is repeated for each guideline definition.

If at a certain point in time some guidelines become incompatible with one another, an additional *incompatible with* part could be added to guideline formulations, and then the algorithm could be modified to assess guideline compatibility as well.

An implementation of this algorithm for the node actions guideline has been created [Jas24]: it is written in Python and makes use of the `lxml` library to parse a TLA$^+$ specification syntax `XML` representation generated by SANY (Syntactic Analyzer) to then discern whether the required definitions are present and the guideline has been applied correctly. There are still limitations to the implementation as it only checks guideline definitions and application properties with the first action bounded by an $\exists$ operator in the $Next$ formula.

This implementation could be further improved and used to implement a Visual Studio Code extension that would highlight certain parts of TLA$^+$ specification files in which the guidelines defined in Section 4.1 could be applied.

It is also worth noting that such an implementation approach would require separate specific algorithms for each guideline to be checked. A possible alternative would be to create a program that could interpret any guideline part expressed in TLA$^+$ syntax as directives to check for specific patterns in a TLA$^+$ specification syntax `XML` representation.

Results

TR1. Original definitions of a synchronous Ben-Or consensus algorithm (Algorithm 2) and an asynchronous Ben-Or consensus algorithm without messages between nodes (the Byzantine variant defined in Algorithm 3, and the normal variant described in Algorithm 5).

TR2. Written Ben-Or TLA$^+$ specifications to achieve the objectives of the master's thesis: `BenOrMsgs`, `BenOrAbstSync`, `BenOrMsgsByz`, `BenOrAbstSyncByz`, `BenOrAbstByz`, `BenOrAbstByzCC`, `BenOrAbstByzCCH`, `BenOrAbstByzCCL`, `BenOrAbstByzCCLH`, `BenOrAbstByzCCLRst`, `BenOrAbstByzCCLRstH`, `BenOrMsgsByzCC`, `BenOrMsgsByzCCH`, `BenOrMsgsByzCCL`, `BenOrMsgsByzCCLH`, `BenOrMsgsByzCCLRst`, and `BenOrMsgsByzCCLRstH` (Figures 23–39 respectively).

TR3. Configuration files for checking temporal properties in state machine models with TLC: `BenOr`, `BenOrByz`, and `BenOrByzCC` (Figures 20–22 respectively). They are an essential part of the results as it is not possible to run TLC on TLA$^+$ specifications mentioned in **TR2** without the definitions present in these configuration files.

TR4. Four defined TLA$^+$ specification writing guidelines for state or behavior space reduction in distributed algorithm TLA$^+$ specifications (Tables 5–12).

TR5. Abstract description of an algorithm to detect specification writing guideline applicability and whether guidelines were applied correctly or not (expressed as a flowchart in Figure 16 with a partial implementation provided in [Jas24]).

TR6. Data collected on how state space size changes in different state machine models defined by written TLA$^+$ specifications with applied specification writing guidelines (shown in Table 4 and in Figures 13, 14, 18, and 19).

TR7. Calculations made on the specification writing guideline state space reduction property SR depicted in Tables 14 and 15.

Conclusions

TC1. Applying the information polling guideline allowed to attain a substantially smaller state space size in TLA$^+$ specifications compared to the information pushing guideline (Table 4 and Figure 14).

TC2. The application of the node actions guideline with a restricted common coin toss action (Figure 9) was able to achieve greater state space reduction as the number of nodes N increased. Conversely, the application of the same guideline with a local unrestricted common coin toss action performed worse than the system actions guideline when $N \geq 8$ (Table 15 and Figure 19).

TC3. Information polling and node actions guidelines were both applied within the `BenOrAbstByzCCLRst` specification to achieve the smallest state space size among all compared TLA$^+$ specifications shown in Table 4.

TC4. Specification writing guidelines were defined with general application cases in mind. In guideline descriptions, it is not stated how a guideline should be applied to a particular specification. Instead, the required definitions and properties that must be true after guideline application are provided.

TC5. Application cases added to guideline formulations are only examples and not definitive ways to apply the presented guidelines. The state space reduction effectiveness of guidelines was shown for only a limited subset of all application cases and therefore there are still many more specific applications of the defined guidelines to consider: the state space reduction of these other application cases might be greater or lower than achieved in Tables 14 and 15.

References

[Akh12] Sabina Akhtar. *Formal Verification of Distributed Algorithms using PlusCal-2*. Dissertation, Université de Lorraine, 2012-05. URL: `https://theses.hal.science/tel-01749162`. URL checked on 2024-05-26.

[Apa24] Apalache. Supported features, 2024-05. URL: `https://apalache.informal.systems/docs/apalache/features.html`. URL checked on 2024-05-26.

[AT12] Marcos K. Aguilera and Sam Toueg. The correctness proof of Ben-Or's randomized consensus algorithm. *Distributed Computing*, 25(5):371–381, 2012-10. ISSN: 1432-0452. DOI: `https://doi.org/10.1007/s00446-012-0162-z`.

[AZT+24] Hamra Afzaal, Nazir Ahmad Zafar, Aqsa Tehseen, Shaheen Kousar and Muhammad Imran. Formal Verification of Justification and Finalization in Beacon Chain. *IEEE Access*, 12:55077–55102, 2024. DOI: `https://doi.org/10.1109/ACCESS.2024.3389551`.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill and L.J. Hwang. Symbolic model checking: 1020 States and beyond. *Information and Computation*, 98(2):142–170, 1992. ISSN: 0890-5401. DOI: `https://doi.org/10.1016/0890-5401(92)90017-A`.

[Bee08] Robert Beers. Pre-RTL Formal Verification: An Intel Experience. *Proceedings of the 45th Annual Design Automation Conference*, p. 806–811, Anaheim, California. Association for Computing Machinery, 2008. ISBN: 9781605581156. DOI: `https://doi.org/10.1145/1391469.1391675`.

[Ben83] Michael Ben-Or. Another Advantage of Free Choice (Extended Abstract): Completely Asynchronous Agreement Protocols. *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, p. 27–30, Montreal, Quebec, Canada. Association for Computing Machinery, 1983. ISBN: 0897911105. DOI: `https://doi.org/10.1145/800221.806707`.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008, p. 975. ISBN: 978-0262026499.

[BKL+21] Nathalie Bertrand, Igor Konnov, Marijana Lazić and Josef Widder. Verification of randomized consensus algorithms under round-rigid adversaries. *International Journal on Software Tools for Technology Transfer*, 23(5):797–821, 2021-10. ISSN: 1433-2787. DOI: `https://doi.org/10.1007/s10009-020-00603-x`.

[CDL+12] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts and Hernán Vanzetto. TLA + Proofs. Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, p. 147–154, Berlin, Heidelberg. Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-32759-9. DOI: `https://doi.org/10.1007/978-3-642-32759-9_14`.

[CGJ+01] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu and Helmut Veith. *Progress on the State Explosion Problem in Model Checking*. *Informatics: 10 Years Back, 10 Years Ahead*. Reinhard Wilhelm, editor. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, p. 176–194. ISBN: 978-3-540-44577-7. DOI: `https://doi.org/10.1007/3-540-44577-3_12`.

[EGK+13] Enver Ever, Orhan Gemikonakli, Altan Kocyigit and Eser Gemikonakli. A hybrid approach to minimize state space explosion problem for the solution of two stage tandem queues. *Journal of Network and Computer Applications*, 36(2):908–926, 2013. ISSN: 1084-8045. DOI: `https://doi.org/10.1016/j.jnca.2012.10.006`.

[EMB24] Javier Esparza-Peidro, Francesc D. Muñoz-Escoí and José M. Bernabéu-Aubán. Modeling microservice architectures. *Journal of Systems and Software*, 213:112041, 2024. ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2024.112041`.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. *SIGPLAN Not.*, 40(1):110–121, 2005-01. ISSN: 0362-1340. DOI: `https://doi.org/10.1145/1047659.1040315`.

[FGH22] Alessandro Fantechi, Stefania Gnesi and Anne E. Haxthausen. Formal Methods for Distributed Control Systems of Future Railways. Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Va-*

lidation. Practice, p. 243–245, Cham. Springer Nature Switzerland, 2022. ISBN: 978-3-031-19762-8. DOI: `https://doi.org/10.1007/978-3-031-19762-8_19`.

[FLP85] Michael J. Fischer, Nancy A. Lynch and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, 1985-04. ISSN: 0004-5411. DOI: `https://doi.org/10.1145/3149.214121`.

[Fou23] Linux Foundation. TLA$^+$ Foundation announcement, 2023-04. URL: `https://www.linuxfoundation.org/press/linux-foundation-launches-tlafoundation`. URL checked on 2024-05-26.

[Git18] Github. Raft consensus algorithm TLA$^+$ specification, 2018-06. URL: `https://github.com/ongardie/raft.tla/blob/master/raft.tla`. URL checked on 2024-05-26.

[Git20] Github. Paxos consensus algorithm TLA$^+$ specification, 2020-09. URL: `https://github.com/tlaplus/Examples/blob/master/specifications/Paxos/Paxos.tla`. URL checked on 2024-05-26.

[GKO15] Jan Friso Groote, Tim W.D.M. Kouters and Ammar Osaiweran. Specification guidelines to avoid the state space explosion problem. *Software Testing, Verification and Reliability*, 25:4–33, 2015. DOI: `https://doi.org/10.1002/stvr.1536`.

[GM12] Stefania Gnesi and Tiziana Margaria. *Formal methods for industrial critical systems: A survey of applications*. John Wiley & Sons, 2012. ISBN: 978-0470876183.

[GM20] Mario Gleirscher and Diego Marmsoler. Formal methods in dependable systems engineering: a survey of professionals from Europe and North America. *Empirical Software Engineering*, 25(6):4473–4546, 2020-11. ISSN: 1573-7616. DOI: `https://doi.org/10.1007/s10664-020-09836-5`.

[Gro24] Google Groups. Inquiry about partial order reduction implementation for TLC, 2024-05. URL: `https://groups.google.com/g/tlaplus/c/7zHA6LPkF3g`. URL checked on 2024-05-26.

[GS12] D. Gentner and L. Smith. Analogical Reasoning. *Encyclopedia of Human Behavior (Second Edition)*, p. 130–136. Academic Press, San Diego, second edition leid., 2012. ISBN: 978-0-08-096180-4. DOI: `https://doi.org/10.1016/B978-0-12-375000-6.00022-7`.

[GtBvdP20] Hubert Garavel, Maurice H. ter Beek and Jaco van de Pol. The 2020 Expert Survey on Formal Methods. Maurice H. ter Beek and Dejan Ničković, editors, *Formal Methods for Industrial Critical Systems*, p. 3–69, Cham. Springer International Publishing, 2020. ISBN: 978-3-030-58298-2. DOI: `https://doi.org/10.1007/978-3-030-58298-2_1`.

[GvdP00] Jan Friso Groote and Jaco van de Pol. State Space Reduction Using Partial τ-Confluence. Mogens Nielsen and Branislav Rovan, editors, *Mathematical Foundations of Computer Science 2000*, p. 383–393, Berlin, Heidelberg. Springer Berlin Heidelberg, 2000. ISBN: 978-3-540-44612-5. DOI: `https://doi.org/10.1007/3-540-44612-5_34`.

[GvdPW23] Mario Gleirscher, Jaco van de Pol and Jim Woodcock. A manifesto for applicable formal methods. *Software and Systems Modeling*, 22(6):1737–1749, 2023-12. ISSN: 1619-1374. DOI: `https://doi.org/10.1007/s10270-023-01124-2`.

[YML99] Yuan Yu, Panagiotis Manolios and Leslie Lamport. Model Checking TLA+ Specifications. Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, p. 54–66, Berlin, Heidelberg. Springer Berlin Heidelberg, 1999. ISBN: 978-3-540-48153-9. DOI: `https://doi.org/10.1007/3-540-48153-2_6`.

[Inr24a] Microsoft Research - Inria. TLA$^+$ proof constructs, 2024-05. URL: `https://tla.msr-inria.inria.fr/tlaps/content/Documentation/Tutorial/Other_proof_constructs.html`. URL checked on 2024-05-26.

[Inr24b] Microsoft Research - Inria. TLAPS practical hints, 2024-05. URL: `https://tla.msr-inria.inria.fr/tlaps/content/Documentation/Tutorial/Practical_hints.html`. URL checked on 2024-05-26.

[Jas24] Marijus Jasinskas. Repository containing the results and other files relevant to the master's thesis, 2024-05. URL: `https://github.com/marjasi/tla_guidelines`. URL checked on 2024-05-26.

[KDL+22] Tomas Kulik, Brijesh Dongol, Peter Gorm Larsen, Hugo Daniel Macedo, Steve Schneider, Peter W. V. Tran-Jørgensen and James Woodcock. A Survey of Practical Formal Methods for Security. *Form. Asp. Comput.*, 34(1), 2022-07. ISSN: 0934-5043. DOI: `https://doi.org/10.1145/3522582`.

[KKM22] Igor Konnov, Markus Kuppe and Stephan Merz. Specification and Verification with the TLA+ Trifecta: TLC, Apalache, and TLAPS. Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles*, p. 88–105, Cham. Springer International Publishing, 2022. ISBN: 978-3-031-19849-6. DOI: `https://doi.org/10.1007/978-3-031-19849-6_6`.

[KKT19] Igor Konnov, Jure Kukovec and Thanh-Hai Tran. TLA+ Model Checking Made Symbolic. *Proc. ACM Program. Lang.*, 3, 2019. DOI: `https://doi.org/10.1145/3360549`.

[KLM+98] R. Kurshan, V. Levin, M. Minea, D. Peled and H. Yenigün. Static partial order reduction. Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, p. 345–357, Berlin, Heidelberg. Springer Berlin Heidelberg, 1998. ISBN: 978-3-540-69753-4. DOI: `https://doi.org/10.1007/BFb0054182`.

[KLS+23] Igor Konnov, Marijana Lazić, Ilina Stoilkovska and Josef Widder. Survey on Parameterized Verification with Threshold Automata and the Byzantine Model Checker. *Logical Methods in Computer Science*, Volume 19, Issue 1, 2023-01. DOI: `https://doi.org/10.46298/lmcs-19(1:5)2023`.

[KMM18] Miika Kalske, Niko Mäkitalo and Tommi Mikkonen. Challenges When Moving from Monolith to Microservice Architecture. Irene Garrigós and Manuel Wimmer, editors, *Current Trends in Web Engineering*, p. 32–47, Cham. Springer International Publishing, 2018. DOI: `https://doi.org/10.1007/978-3-319-74433-9_3`.

[Lam01] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*:51–58, 2001-12. URL: `https://www.microsoft.com/en-us/research/publication/paxos-made-simple`. URL checked on 2024-05-26.

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 2002, p. 384. ISBN: 978-0321143068.

[LCL87] F. J. Lin, P. M. Chu and M. T. Liu. Protocol Verification Using Reachability Analysis: The State Space Explosion Problem and Relief Strategies. *SIGCOMM Comput. Commun. Rev.*, 17(5):126–135, 1987-08. ISSN: 0146-4833. DOI: `https://doi.org/10.1145/55483.55496`.

[LM22] Leslie Lamport and Stephan Merz. Prophecy Made Simple. *ACM Trans. Program. Lang. Syst.*, 44(2), 2022-04. ISSN: 0164-0925. DOI: `https://doi.org/10.1145/3492545`.

[mCR24] mCRL2. 2024. URL: `https://www.mcrl2.org/web/index.html`. URL checked on 2024-05-26.

[New14] Chris Newcombe. Why Amazon Chose TLA +. *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, p. 25–39, Berlin, Heidelberg. Springer Berlin Heidelberg, 2014. ISBN: 978-3-662-43652-3. DOI: `https://doi.org/10.1007/978-3-662-43652-3_3`.

[NGY+21] Faranak Nejati, Abdul Azim Abd Ghani, Ng Keng Yap and Azmi Bin Jafaar. Handling State Space Explosion in Component-Based Software Verification: A Review. *IEEE Access*, 9:77526–77544, 2021. DOI: `https://doi.org/10.1109/ACCESS.2021.3081742`.

[Nic11] Eóin Woods Nick Rozanski. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, 2011, p. 704. ISBN: 978-0321718334.

[NRZ+15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Commun. ACM*, 58(4):66–73, 2015. ISSN: 0001-0782. DOI: `https://doi.org/10.1145/2699417`.

[OKK+23] Rodrigo Otoni, Igor Konnov, Jure Kukovec, Patrick Eugster and Natasha Sharygina. Symbolic Model Checking for TLA+ Made Faster. Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, p. 126–144, Cham. Springer Nature Switzerland, 2023. ISBN: 978-3-031-30823-9. DOI: `https://doi.org/10.1007/978-3-031-30823-9_7`.

[OO14] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, p. 305–320, Philadelphia, PA. USENIX Association, 2014. ISBN: 9781931971102. URL: `https://dl.acm.org/doi/10.5555/2643634.2643666`. URL checked on 2024-05-26.

[PAH+08] P.W. C. Prasad, A. Assi, A. Harb and V.C. Prasad. Binary Decision Diagrams: An Improved Variable Ordering using Graph Representation of Boolean Functions. Version 6093, 2008-02. DOI: `https://doi.org/10.5281/zenodo.1062260`.

[Pel09] Radek Pelánek. Fighting State Space Explosion: Review and Evaluation. Darren Cofer and Alessandro Fantechi, editors, *Formal Methods for Industrial Critical Systems*, p. 37–52, Berlin, Heidelberg. Springer Berlin Heidelberg, 2009. ISBN: 978-3-642-03240-0. DOI: `https://doi.org/10.1007/978-3-642-03240-0_7`.

[Pro24] Gregory Provan. Formal Methods for Autonomous Vehicles. *IT Professional*, 26(1):50–56, 2024. DOI: `https://doi.org/10.1109/MITP.2024.3356158`.

[Rus24] Daniel Russo. Navigating the Complexity of Generative AI Adoption in Software Engineering. *ACM Trans. Softw. Eng. Methodol.*, 2024-03. ISSN: 1049-331X. DOI: `https://doi.org/10.1145/3652154`.

[Val98] Antti Valmari. *The state explosion problem*. *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*. Wolfgang Reisig and Grzegorz Rozenberg, editors. Springer

Berlin Heidelberg, Berlin, Heidelberg, 1998, p. 429–528. ISBN: 978-3-540-49442-3. DOI: https://doi.org/10.1007/3-540-65306-6_21.

[VBF+11] Erics Verhulst, Raymond T. Boute, José Miguel Sampaio Faria, Bernhard H. C. Sputh and Vitaliy Mazhuyev. *Formal Development of a Network-Centric RTOS: Software Engineering for Reliable Embedded Systems*. Springer, 2011, p. 236. ISBN: 978-1441997357. DOI: https://doi.org/10.1007/978-1-4419-9736-4.

[Vel19] Daniel J. Velleman. *How to Prove It: A Structured Approach, 3rd Edition*. Cambridge University Press, 2019, p. 468. ISBN: 978-1108439534.

[vGW96] Rob J. van Glabbeek and W. Peter Weijland. Branching Time and Abstraction in Bisimulation Semantics. *J. ACM*, 43(3):555–600, 1996. ISSN: 0004-5411. DOI: https://doi.org/10.1145/233551.233556.

[Woh14] C. Wohlin. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE 2014)*, p. 321–330, London, UK, 2014. URL: https://www.wohlin.eu/ease14.pdf. URL checked on 2024-05-26.

[ZZW+24] Jianyu Zhang, Long Zhang, Yixuan Wu and Feng Yang. An Agile Formal Specification Language Design Based on K Framework, 2024. DOI: https://doi.org/10.48550/arXiv.2404.18515.

Terms and definitions

- **Action** – A formula that contains both primed and unprimed variables [Lam02].
- **Behavior** – A sequence of states [Lam02].
- **Common coin** – A conceptual tool in distributed systems used to model shared randomness. It represents an external event that affects multiple processes simultaneously [AT12].
- **Deadlock** – A state in which the action $Next$ is disabled [Lam02].
- **Disabled action** – An action Act is disabled when it is not possible to take a step Act from the current state [Lam02].
- **Enabled action** – An action Act is enabled if and only if it is possible to take a step Act from the current state [Lam02].
- **Global action scope** – An aspect of an action to not contain any parameters and to be treated as taken by the system in distributed algorithm TLA$^+$ specifications.
- **Information polling** – Modeling of node communication so that nodes can obtain required internal values from other nodes without them being shared via messages.
- **Information pushing** – Modeling of node communication so that internal values have to be sent via messages for them to be accessible to other nodes.
- **Invariant** – A state predicate that is true in all reachable states [Lam02].
- **Liveness property** – A property that asserts that something must eventually happen in the system, expressed as a temporal formula [Lam02].
- **Local action scope** – An aspect of an action to contain a parameter that is a node identifier and to be treated as taken by a specific node in distributed algorithm TLA$^+$ specifications.
- **Model checking** – A formal verification technique used to check whether a given model satisfies desired properties by exhaustively exploring all possible system states [BK08].
- **Over-abstraction** – A situation where a system model or specification includes excessively simplified details, leading to potential incorrect conclusions after its analysis.
- **Primed variable** – A variable's value in the next state denoted with the prime ($'$) symbol [Lam02].
- **Safety property** – A property that asserts that something can not occur in the system, expressed as a temporal formula [Lam02].

- **State** – An assignment of values to defined variables [Lam02].
- **Step** – A pair of successive states in a behavior [Lam02].
- **Stuttering step** – A step during which the values of defined variables are unchanged [Lam02].
- **Temporal property** – A safety or liveness property [Lam02].
- **Unprimed variable** – A variable’s value in the current state [Lam02].

Abbreviations

- **mCRL2** – micro Common Representation Language 2; a formal specification language for describing concurrent discrete event systems founded by Jan Friso Groote [mCR24].
- **SANY** – Syntactic Analyzer; a parser and syntax checker for TLA+ specifications [Lam02].
- **TLA** – Temporal Logic of Actions; a logic developed by Leslie Lamport and used as the basis for TLA$^+$ [Lam02].
- **TLA$^+$** – Temporal Logic of Actions$^+$; a formal specification language developed by Leslie Lamport [Lam02].
- **TLAPS** – TLA$^+$ Proof System; a system for mechanically checking proofs written in TLA+ [Lam02].
- **TLC** – an explicit-state model checker for TLA$^+$ specifications [Lam02].

Appendix 1

Additional material

```
——————————————— MODULE Simple ———————————————
EXTENDS TLC, FiniteSets, Integers

CONSTANT NODES

VARIABLE msgs
VARIABLE proposal
VARIABLE decision

vars ≜ ⟨msgs, proposal, decision⟩

ProposalValues ≜ {"0", "1"}
DecisionValues ≜ ProposalValues ∪ {"-1"}  -1 is used for initial decision values.

Message ≜ [sender : NODES, value : ProposalValues]
Proposals  ≜ [NODES → ProposalValues]
Decisions  ≜ [NODES → DecisionValues]

Max(a, b) ≜
        IF a > b
         THEN a
         ELSE  b

SendProposal(node) ≜
    ∧ LET messageRecord ≜ [sender ↦ node, value ↦ proposal[node]]
       IN    ∧ messageRecord ∉ msgs
             ∧ msgs' = msgs ∪ {[sender ↦ node, value ↦ proposal[node]]}
    ∧ UNCHANGED ⟨proposal, decision⟩

Decide(node) ≜
    ∧ Cardinality(msgs) ∗ 2 > Cardinality(NODES)  Received more than N/2 messages.
    ∧ LET ZeroOccurence ≜ Cardinality({msg ∈ msgs : msg.value = "0"})
          OneOccurence ≜ Cardinality({msg ∈ msgs : msg.value = "1"})
       IN   decision' = [decision EXCEPT ![node] = Max(ZeroOccurence, OneOccurence)]
    ∧ UNCHANGED ⟨msgs, proposal⟩

TypeOK ≜
    ∧    msgs ⊆ Message
    ∧    proposal ⊆ Proposals
    ∧    decision ⊆ Decisions

Init ≜
    ∧ msgs = {}
    ∧ proposal = [node ∈ NODES ↦ "0"]  All nodes propose the value "0".
    ∧ decision = [node ∈ NODES ↦ "-1"]  All initial decision values are "-1".

Next ≜
   ∃ node ∈ NODES :
      ∨ SendProposal(node)
    ∨ Decide(node)

Spec ≜ Init ∧ □[Next]_vars

THEOREM TypeOKProof ≜ Spec ⇒ □TypeOK
THEOREM DecisionsMade ≜ Spec ⇒ ◇(∀ node ∈ NODES : decision[node] ∈ ProposalValues)
```

Figure 17. A TLA⁺ specification used for the τ-confluence state space reduction example

Algorithm 4 Byzantine Ben-Or consensus algorithm for process p by Michael Ben-Or [Ben83], based on Algorithm 1

```
Input:
    $f \geq 1;\ f < N/5;\ N \geq 6;$
    $rnd = 0;\ est_p \in \{0,1\};\ rval \in \{0,1\};\ pval \in \{?,0,1\};\ dec_p \notin \{0,1\}$
Output:
    All non-faulty processes decide on the same decision value $dec_p \in \{0,1\}$
1: while true do
2:     $rnd = rnd + 1$                                                    \\ Current round
3:     send $(Report, rnd, est_p)$ to all processes                       \\ Phase 1
4:     wait for $N - f$ $(Report, rnd, rval)$ messages
5:     if received more than $(N+f)/2$ $Report$ messages with the same value $val$ then
6:         send $(Proposal, rnd, val)$ to all processes
7:     else
8:         send $(Proposal, rnd, ?)$ to all processes
9:     end if
10:    wait for $N - f$ $(Proposal, rnd, pval)$ messages                  \\ Phase 2
11:    if received at least $(N+f)/2$ $Proposal$ messages with the same value $val \neq ?$ then
12:        $dec_p = val$
13:    end if
14:    if received at least $f+1$ $Proposal$ messages with the same value $val \neq ?$ then
15:        $est_p = val$
16:    else
17:        $est_p$ = random value from $\{0,1\}$
18:    end if
19: end while
```

Algorithm 5 Asynchronous Ben-Or consensus algorithm without message sending for process p, based on Algorithm 2

```
Input:
    $f \geq 1;\ f < N/2;\ N \geq 4;$
    $Rounds = \mathbb{N};$
    $\forall\ round \geq 2 \in Rounds : est[round]_p \notin \{0,1\};$
    $\forall\ round \geq 1 \in Rounds : proposalValue[round]_p \notin \{?,0,1\};$
    $rnd = 1;\ est[1]_p \in \{0,1\};\ dec_p \notin \{0,1\}$
Output:
    All alive processes decide on the same decision value $dec_p \in \{0,1\}$
 1: while true do
 2:     wait until for at least $N - f$ other processes $p'$: $est[rnd]_{p'} \in \{0,1\}$
 3:     if for more than $N/2$ other processes $p'$: $est[rnd]_{p'} = val$ then
 4:         $proposalValue[rnd]_p = val$
 5:     else
 6:         $proposalValue[rnd]_p = ?$
 7:     end if                                      \\ End of phase 1 for process $p$
 8:     wait until for at least $N - f$ other processes $p'$: $proposalValue[rnd]_{p'} \in \{?,0,1\}$
 9:     if for at least $f + 1$ other processes $p'$: $proposalValue[rnd]_{p'} = val$ and $val \neq ?$ then
10:         $dec_p = val$
11:     end if
12:     if for at least one other process $p'$: $proposalValue[rnd]_{p'} = val$ and $val \neq ?$ then
13:         $est[rnd]_p = val$
14:     else
15:         $est[rnd]_p =$ random value from $\{0,1\}$
16:     end if
17:     $est[rnd + 1]_p = est[rnd]_p$
18:     $rnd = rnd + 1$                              \\ End of phase 2 for process $p$
19: end while
```

$$
\begin{aligned}
&IF\ CHECK\_ALL\_INITIAL\_VALUES\\
&THEN\ \exists\ initialEstimates \in AllEstimates :\\
&\quad estimatesAtRound = [round \in NodeRounds\ \mapsto\\
&\qquad\qquad\qquad\qquad IF\ round = 1\\
&\qquad\qquad\qquad\qquad THEN\ initialEstimates\\
&\qquad\qquad\qquad\qquad ELSE\ [node \in NODES\ \mapsto\ \text{“}-1\text{”}]]\\
&ELSE\ estimatesAtRound = [round \in NodeRounds\ \mapsto\\
&\qquad\qquad\qquad\qquad IF\ round = 1\\
&\qquad\qquad\qquad\qquad THEN\ [node \in NODES\ \mapsto\\
&\qquad\qquad\qquad\qquad\quad IF\ node \in ESTIMATE\_ZERO\\
&\qquad\qquad\qquad\qquad\quad THEN\ \text{“0”}\\
&\qquad\qquad\qquad\qquad\quad ELSE\ \text{“1”}]\\
&\qquad\qquad\qquad\qquad ELSE\ [node \in NODES\ \mapsto\ \text{”}-1\text{”}]]
\end{aligned}
\tag{22}
$$

$$
\begin{aligned}
&\exists\, round \in NodeRounds :\\
&\quad \land\, commonCoin[round] = \text{“}{-1}\text{”}\\
&\quad \land \lor \exists\, node \in NODES \setminus FAULTY\_NODES :\\
&\qquad\qquad \land\, estimatesAtRound[round][node] \neq \text{“}{-1}\text{”}\\
&\qquad\qquad \land\, WaitForProposals(node)\\
&\qquad\qquad \land \lor CheckPhase2a(node) \land CheckPhase2b(node)\\
&\qquad\qquad\quad \lor \neg CheckPhase2a(node) \land CheckPhase2b(node)\\
&\qquad\qquad \land\, commonCoin' =\\
&\qquad\qquad\qquad [commonCoin\ EXCEPT\ ![round] =\\
&\qquad\qquad\qquad\qquad estimatesAtRound[round][node]]\\
&\qquad \lor\ \land \forall\, node \in NODES \setminus FAULTY\_NODES :\\
&\qquad\qquad \land\, WaitForProposals(node)\\
&\qquad\qquad \land\, \neg CheckPhase2a(node)\\
&\qquad\qquad \land\, \neg CheckPhase2b(node)\\
&\qquad\quad \land\ \exists\, coinValue \in DecisionValues :\\
&\qquad\qquad commonCoin' =\\
&\qquad\qquad\qquad [commonCoin\ EXCEPT\ ![round] = coinValue]
\end{aligned}
\tag{23}
$$

$$
\begin{aligned}
&Next \triangleq\\
&\quad \land\, \neg LastState\\
&\quad \land \lor \exists\, node \in NODES :\\
&\qquad\qquad \lor\ Phase1(node)\\
&\qquad\qquad \lor\ Phase2(node)\\
&\qquad\qquad \lor\ SetNextRoundEstimate(node)\\
&\qquad \lor\ TossCommonCoin
\end{aligned}
\tag{24}
$$

$$
\begin{aligned}
&Spec \triangleq \land\ Init\\
&\quad \land \Box[Next]\_vars\\
&\quad \land \forall\ node \in NODES :\\
&\qquad\qquad WF\_vars(Phase1(node) \lor Phase2(node) \lor SetNextRoundEstimate(node))\\
&\quad \land WF\_vars(TossCommonCoin)
\end{aligned}
\tag{25}
$$

$$
\begin{aligned}
&ConsensusReachedByzantine \triangleq\\
&\quad \land \forall\ node \in NODES \setminus FAULTY\_NODES : decisions[node] \in DecisionValues\\
&\quad \land \forall\ node1,\ node2 \in NODES \setminus FAULTY\_NODES : decisions[node1] = decisions[node2]\\
&ConsensusProperty \triangleq \Diamond\Box[ConsensusReachedByzantine]\_vars
\end{aligned}
\tag{26}
$$

$$
\begin{aligned}
&TossCommonCoin(settingNode) \triangleq \\
&\quad \land\ commonCoin[rounds[settingNode]] = \text{“}-1\text{”} \\
&\quad \land \lor \exists\ node \in NODES \setminus FAULTY\_NODES : \\
&\qquad\qquad \land\ estimatesAtRound[rounds[settingNode]][node] \neq \text{“}-1\text{”} \\
&\qquad\qquad \land\ WaitForProposals(node) \\
&\qquad\qquad \land \lor CheckPhase2a(node) \land CheckPhase2b(node) \\
&\qquad\qquad\quad \lor \neg CheckPhase2a(node) \land CheckPhase2b(node) \\
&\qquad\qquad \land\ commonCoin' = \\
&\qquad\qquad\quad [commonCoin\ EXCEPT\ ![rounds[settingNode]] = \\
&\qquad\qquad\qquad\qquad estimatesAtRound[rounds[settingNode]][node]] \\
&\qquad \lor\ \land \forall\ node \in NODES \setminus FAULTY\_NODES : \\
&\qquad\qquad \land\ WaitForProposals(node) \\
&\qquad\qquad \land \neg CheckPhase2a(node) \\
&\qquad\qquad \land \neg CheckPhase2b(node) \\
&\qquad\quad \land\ \exists\ coinValue \in DecisionValues : \\
&\qquad\qquad commonCoin' = \\
&\qquad\qquad\quad [commonCoin\ EXCEPT\ ![rounds[settingNode]] = coinValue]
\end{aligned}
\tag{27}
$$

$$
\begin{aligned}
&Next \triangleq \\
&\quad \exists\ node \in NODES : \\
&\qquad \land \neg LastState \\
&\qquad \land \lor Phase1(node) \\
&\qquad\quad \lor\ Phase2(node) \\
&\qquad\quad \lor\ TossCommonCoin(node) \\
&\qquad\quad \lor\ SetNextRoundEstimate(node)
\end{aligned}
\tag{28}
$$

$$
\begin{aligned}
&Spec \triangleq \land\, Init \\
&\quad \land\, \Box[Next]\_vars \\
&\quad \land\, \forall\, node \in NODES : \\
&\qquad WF\_vars(\lor\, Phase1(node) \,\lor\, Phase2(node) \\
&\qquad\qquad\qquad \lor\, WF\_vars(TossCommonCoin(node)) \,\lor\, SetNextRoundEstimate(node))
\end{aligned}
\tag{29}
$$

$$
\begin{aligned}
&TossCommonCoin(settingNode) \triangleq \\
&\quad \land\, settingNode = CCNode \\
&\quad \land\, commonCoin[rounds[settingNode]] = \text{“}{-1}\text{”} \\
&\quad \land\, \lor\, \exists\, node \in NODES \setminus FAULTY\_NODES : \\
&\qquad\qquad \land\, estimatesAtRound[rounds[settingNode]][node] \;\neq\; \text{“}{-1}\text{”} \\
&\qquad\qquad \land\, WaitForProposals(node) \\
&\qquad\qquad \land\, \lor\, CheckPhase2a(node) \land CheckPhase2b(node) \\
&\qquad\qquad\quad \lor\, \neg CheckPhase2a(node) \land CheckPhase2b(node) \\
&\qquad\qquad \land\, commonCoin' = \\
&\qquad\qquad\quad [commonCoin\ EXCEPT\ ![rounds[settingNode]] = \\
&\qquad\qquad\qquad estimatesAtRound[rounds[settingNode]][node]] \\
&\qquad \lor\, \land \forall\, node \in NODES \setminus FAULTY\_NODES : \\
&\qquad\qquad \land\, WaitForProposals(node) \\
&\qquad\qquad \land\, \neg CheckPhase2a(node) \\
&\qquad\qquad \land\, \neg CheckPhase2b(node) \\
&\qquad \land\, \exists\, coinValue \in DecisionValues : \\
&\qquad\qquad commonCoin' = \\
&\qquad\qquad\quad [commonCoin\ EXCEPT\ ![rounds[settingNode]] = coinValue]
\end{aligned}
\tag{30}
$$

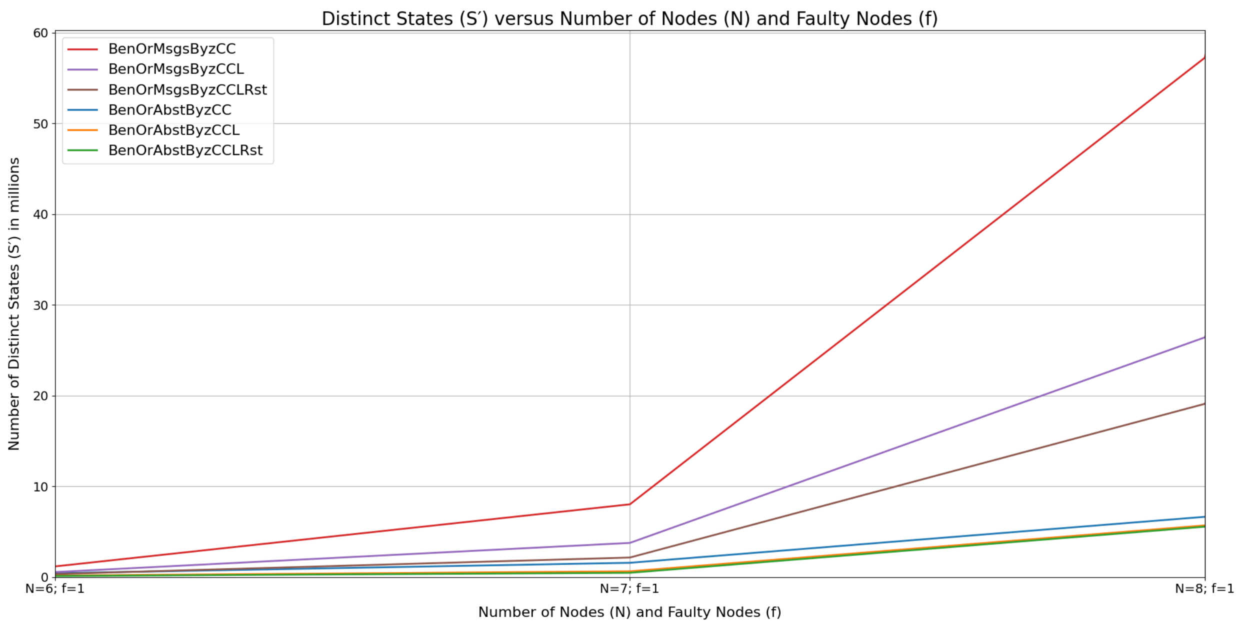

Figure 18. Distinct state space when $N \leq 8$ diagram from data in Table 4

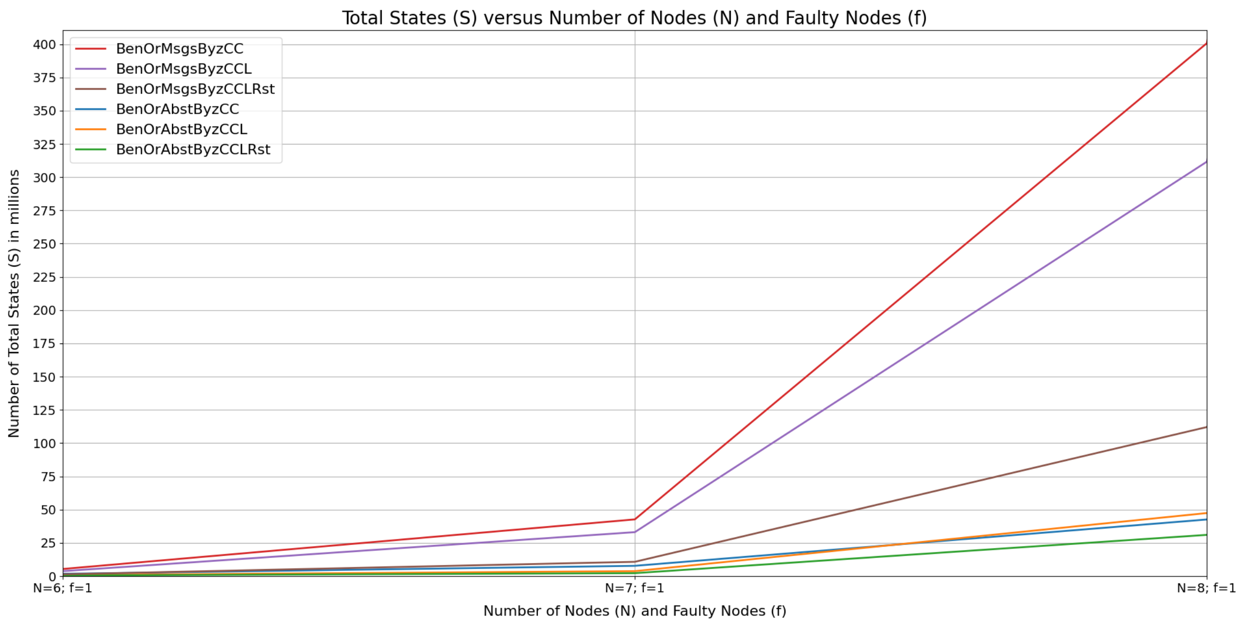

Figure 19. Total state space when $N \leq 8$ diagram from data in Table 4

$$
\begin{aligned}
&RefMsgsH \triangleq INSTANCE\ BenOrMsgsByzCCLRstH \\
&\qquad\qquad WITH\ msgs \leftarrow msgsHistory, \\
&\qquad\qquad\qquad estimates \leftarrow estimateHistory \\
&\qquad\qquad\qquad estimateHistory \leftarrow estimatesAtRound \\
&\qquad\qquad\qquad proposalHistory \leftarrow proposalsAtRound \\
&RefinementProperty \triangleq RefMsgsH!Spec
\end{aligned}
\tag{31}
$$

$$
\begin{aligned}
&RefAbstH \triangleq INSTANCE\ BenOrMsgsByzCCLRstH \\
&\qquad\qquad WITH\ msgsHistory \leftarrow msgs, \\
&\qquad\qquad\qquad estimateHistory \leftarrow estimates \\
&\qquad\qquad\qquad estimatesAtRound \leftarrow estimateHistory \\
&\qquad\qquad\qquad proposalsAtRound \leftarrow proposalHistory \\
&RefinementProperty \triangleq RefAbstH!Spec
\end{aligned}
\tag{32}
$$

Appendix 2

Ben-Or TLA$^+$ specifications

Table 16. Properties verified by TLC when model checking written Ben-Or TLA$^+$ specifications; CN – $ConsensusProperty$, CC – $CommonCoinProperty$, and REF – $RefinementProperty$

Nodes	BenOrMsgsByz						BenOrAbstByz					
	CC		CCL		CCLRst		CC		CCL		CCLRst	
	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness
N=6; f=1	TypeOK	CN, CC	TypeOK	CN, CC, REF	TypeOK	CN, CC, REF	TypeOK	CN, CC	TypeOK	CN, CC, REF	TypeOK	CN, CC, REF
N=7; f=1	TypeOK	CN, CC	TypeOK	CN, CC, REF	TypeOK	CN, CC, REF	TypeOK	CN, CC	TypeOK	CN, CC, REF	TypeOK	CN, CC, REF
N=8; f=1	TypeOK	CN, CC	TypeOK	CN, CC, REF	TypeOK	CN, CC, REF	TypeOK	CN, CC	TypeOK	CN, CC, REF	TypeOK	CN, CC, REF
N=9; f=1	TypeOK	CN, CC	TypeOK	CN, CC, REF	TypeOK	CN, CC, REF	TypeOK	CN, CC	TypeOK	CN, CC, REF	TypeOK	CN, CC, REF

```
CONSTANTS
   NODES = {"n1", "n2", "n3", "n4"}   \* The set of nodes.
   FAULTY_NODES = {"n1"}              \* Which nodes are faulty.
   DECISION_ZERO = {"n1", "n2"}       \* Which nodes decide zero initially.
   DECISION_ONE = {"n3", "n4"}        \* Which nodes decide one initially.
   CHECK_ALL_INITIAL_VALUES = TRUE    \* Check all possible initial estimates.

CHECK_DEADLOCK FALSE \* Must be TRUE if using LastState formula to stop TLC.

SPECIFICATION Spec

INVARIANTS
    TypeOK

PROPERTIES
    ConsensusProperty
```

Figure 20. BenOr.cfg configuration file used for TLA$^+$ specifications of the Ben-Or consensus algorithm

```
CONSTANTS
    NODES = {"n1", "n2", "n3", "n4", "n5", "n6"}
    FAULTY_NODES = {"n1"}
    ESTIMATE_ZERO = {"n1", "n2"}
    ESTIMATE_ONE = {"n3", "n4", "n5", "n6"}
    CHECK_ALL_INITIAL_VALUES = TRUE

CHECK_DEADLOCK FALSE \* Must be TRUE if using LastState formula to stop TLC.

SPECIFICATION Spec

INVARIANTS
    TypeOK

PROPERTIES
    ConsensusProperty
```

Figure 21. BenOrByz.cfg configuration file used for TLA$^+$ specifications of the Byzantine version of the Ben-Or consensus algorithm

```
CONSTANTS
    NODES = {"n1", "n2", "n3", "n4", "n5", "n6"}
    FAULTY_NODES = {"n1"}
    ESTIMATE_ZERO = {"n1", "n2"}
    ESTIMATE_ONE = {"n3", "n4", "n5", "n6"}
    CHECK_ALL_INITIAL_VALUES = TRUE

CHECK_DEADLOCK FALSE \* Must be TRUE if using LastState formula to stop TLC.

SPECIFICATION Spec

INVARIANTS
    TypeOK

PROPERTIES
    ConsensusProperty
    CommonCoinProperty
    RefinementProperty \* Can be commented out when not checking refinement.
```

Figure 22. BenOrByzCC.cfg configuration file used for TLA^{+} specifications of the Byzantine version of the Ben-Or consensus algorithm that use the common coin

MODULE *BenOrMsgs*

This is a specification of the *BenOr* consensus algorithm (not the *Byzantine* version). In the specification, messages are added to the *msgs* set and message records contain
all necessary values for communication between nodes.
Compared to the abstract specification, this is a more detailed specification with a lot bigger state space size. As the number of rounds *NodeRounds* is an infinite set, new states are always reachable until consensus is achieved.

EXTENDS TLC, $Integers$, $FiniteSets$

Use the *BenOr.cfg* file.

CONSTANTS $NODES$, $FAULTY\_NODES$, $ESTIMATE\_ZERO$, $ESTIMATE\_ONE$, $CHECK\_ALL\_INITIAL\_VALUES$

VARIABLE *rounds* Function that returns a given node's current round.
VARIABLE *estimates* Function that returns a given node's current estimated value.
VARIABLE *decisions* Function that returns a given node's current decision value.
VARIABLE *msgs* Set of all sent messages.

$vars \triangleq \langle rounds, estimates, decisions, msgs \rangle$

F must be greater than 0, otherwise the $N - F$ check for proposal messages won't work.

ASSUME $\land (Cardinality(NODES) \geq 2 * Cardinality(FAULTY\_NODES) + 1)$
$\quad\quad\land (Cardinality(FAULTY\_NODES) \geq 1)$

Nodes and their decisions must be specified.

ASSUME $\land NODES \neq \{\}$
$\quad\quad\land ESTIMATE\_ONE \neq \{\}$
$\quad\quad\land ESTIMATE\_ZERO \neq \{\}$

Decision values must be specified for all nodes.

ASSUME $Cardinality(ESTIMATE\_ZERO) + Cardinality(ESTIMATE\_ONE) = Cardinality(NODES)$

Numbers of nodes and faulty nodes.

$NumberOfNodes \triangleq Cardinality(NODES)$
$NumberOfFaultyNodes \triangleq Cardinality(FAULTY\_NODES)$

Set of node round numbers.

$NodeRounds \triangleq Nat$

Decision values of binary consensus.

$DecisionValues \triangleq \{\text{"0"}, \text{"1"}\}$

Set of all possible node decision values including the "-1" used for the initial state.

$NodeDecisionValues \triangleq DecisionValues \cup \{\text{"-1"}\}$

Set of all functions that map nodes to their estimate values.

$NodeEstimateFunctions \triangleq [NODES \rightarrow DecisionValues]$

Set of all possible node estimate values. The "-1" is not necessary since all nodes estimate a value of "0" or "1" in the initial state.

$NodeEstimateValues \triangleq DecisionValues$

Possible values and types of messages in the *BenOr* consensus algorithm.

$MessageType \triangleq \{\text{"report"}, \text{"proposal"}\}$
$MessageValues \triangleq \{\text{"0"}, \text{"1"}, \text{"?"}\}$

Each message record also has a round number.

$MessageRecordSet \triangleq [Sender : NODES, Round : Nat, Type : MessageType, Value : MessageValues]$

Used to print variable values. Must be used with $\land$ and in states which are reached by *TLC*.

$PrintVal(message, expression) \triangleq Print(\langle message, expression \rangle, \text{TRUE})$

A set of all nodes other than the specified node.

$AllOtherNodes(node) \triangleq$
$\quad NODES \setminus \{node\}$

Returns TRUE if a node sent the specified message during the specified round.

$DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue) \triangleq$
$\quad$ LET $messageRecord \triangleq [Sender \mapsto sendingNode, Round \mapsto nodeRound, Type \mapsto messageType, Value \mapsto messageValue]$
$\quad$ IN $\quad messageRecord \in msgs$

Returns TRUE if a node sent a message of the specified type with any value ("0", "1", or "?") during the specified round.

```
$DidNodeSendMessageOfType(sendingNode, nodeRound, messageType) \triangleq$
    $\exists\, msgValue \in MessageValues :$
        LET $messageRecord \triangleq [Sender \mapsto sendingNode, Round \mapsto nodeRound, Type \mapsto messageType, Value \mapsto msgValue]$
        IN  $messageRecord \in msgs$
```

Adds a specific message to the set *msgs* with the sending node and the node's round. If the a message with the same value was already sent during the specified round, it is not added to
the *msgs* set.

```
$SendBroadcastMessage(sendingNode, nodeRound, messageType, messageValue) \triangleq$
    $\land \neg DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue)$
    $\land msgs' = msgs \cup \{[Sender \mapsto sendingNode, Round \mapsto nodeRound, Type \mapsto messageType, Value \mapsto messageValue]\}$
```

Returns the number of messages with the specified value that were sent by nodes other than *sendingNode*.

```
$HowManyMessages(sendingNode, nodeRound, messageType, messageValue) \triangleq$
    $Cardinality(\{msg \in msgs :$
                    $\land msg.Sender \neq sendingNode$
                    $\land msg.Round = nodeRound$
                    $\land msg.Type = messageType$
                    $\land msg.Value = messageValue\})$
```

Returns the number of messages sent by nodes other than *checkingNode* of the specified message type.

```
$HowManyMessagesOfType(checkingNode, nodeRound, messageType) \triangleq$
    $Cardinality(\{node \in AllOtherNodes(checkingNode) :$
        $DidNodeSendMessageOfType(node, nodeRound, messageType)\})$
```

Returns a set of estimate values that are present in more than N / 2 report messages. Only one same value can be present in more than N / 2 messages.

```
$MajorityOfReports(node, nodeRound) \triangleq$
    $\{estimateValue \in NodeEstimateValues :$
        $HowManyMessages(node, nodeRound, \text{“report”}, estimateValue) * 2 > NumberOfNodes\}$
```

Check if there are more than N / 2 report messages with the same value.

```
$CheckPhase1(node, nodeRound) \triangleq$
    $MajorityOfReports(node, nodeRound) \neq \{\}$
```

After each node sends a report message during the first phase, it waits for $N - F$ report messages and then counts how many messages contain the same estimate value v.
If there are more than N / 2 of such messages, the node sends a proposal message with the value that was in more than N / 2 report messages.
Otherwise, the node sends a proposal message with the "?" value.

```
$WaitForReports(node) \triangleq$
    $\land node \in NODES$
    $\land DidNodeSendMessage(node, rounds[node], \text{“report”}, estimates[node])$
    $\land HowManyMessagesOfType(node, rounds[node], \text{“report”}) \geq NumberOfNodes - NumberOfFaultyNodes$
    $\land$ IF $CheckPhase1(node, rounds[node])$
        THEN $\exists\, majorityValue \in MajorityOfReports(node, rounds[node]) :$
                  $SendBroadcastMessage(node, rounds[node], \text{“proposal”}, majorityValue)$
        ELSE $SendBroadcastMessage(node, rounds[node], \text{“proposal”}, \text{“?”})$
    $\land$ UNCHANGED $\langle rounds, estimates, decisions\rangle$
```

The first phase of the *BenOr* consensus algorithm. All nodes send report messages with their estimate value v to all other nodes. Each node then checks what kind of values they have received from other nodes. The value v that is sent as a proposal message if it is in the majority of report message
is not necessarily the same as the node's current estimate value. A node can send a value different from its own estimate value to other nodes in a proposal message.

```
$Phase1(node) \triangleq$
    $\land node \in NODES$
    $\land \lor SendBroadcastMessage(node, rounds[node], \text{“report”}, estimates[node])$
      $\lor WaitForReports(node)$
    $\land$ UNCHANGED $\langle rounds, estimates, decisions\rangle$
```

Returns a set of proposal values "0" or "1" that are proposed by at least $F + 1$ nodes.

```
$AtLeastNF1Proposals(node) \triangleq$
    $\{estimateValue \in NodeEstimateValues :$
        $HowManyMessages(node, rounds[node], \text{“proposal”}, estimateValue) \geq NumberOfFaultyNodes + 1\}$
```

Check for $F + 1$ or more proposal messages with the same value.

```
$CheckPhase2a(node) \triangleq$
    $AtLeastNF1Proposals(node) \neq \{\}$
```

Returns a set of proposal values "0" or "1" that have been proposed by at least one node.

```
$AtLeast1Proposal(node) \triangleq$
    $\{estimateValue \in NodeEstimateValues :$
        $HowManyMessages(node, rounds[node], \text{"proposal"}, estimateValue) \geq 1\}$
```

Check for at least one proposal of a node without the "?" value.

```
$CheckPhase2b(node) \triangleq$
    $AtLeast1Proposal(node) \neq \{\}$
```

Decide an estimate value that has been proposed by at least $F + 1$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$DecidePhase2a(node) \triangleq$
    $\exists\, decisionValue \in AtLeastNF1Proposals(node) :$
        $decisions' = [decisions\ \text{EXCEPT}\ ![node] = decisionValue]$
```

Estimate a proposal value that has been proposed by at least one node. If there are multiple such values, *TLC* will check all possible choices separately.

```
$EstimatePhase2b(node) \triangleq$
    $\exists\, estimateValue \in AtLeast1Proposal(node) :$
        $estimates' = [estimates\ \text{EXCEPT}\ ![node] = estimateValue]$
```

Pick a random estimate value from "0" or "1" for a node. *TLC* will check all possible choices separately.

```
$EstimatePhase2c(node) \triangleq$
    $\exists\, randomEstimate \in NodeEstimateValues :$
        $\land\ estimates' = [estimates\ \text{EXCEPT}\ ![node] = randomEstimate]$
```

The second phase of the *BenOr* consensus algorithm. All nodes that have sent a proposal message wait for $N - F$ of such messages from other nodes. If the node receives at least $F + 1$ proposal messages with the same value v ("0" or "1"), then
the node sets its decision value to v. Furthermore, if the node receives at least one proposal message with the value v, it sets its estimate value to v. Otherwise, the node changes its estimate value to "0" or "1" randomly.

```
$Phase2(node) \triangleq$
    $\land\ node \in NODES$
    $\land\ DidNodeSendMessageOfType(node, rounds[node], \text{"proposal"})$
    $\land\ HowManyMessagesOfType(node, rounds[node], \text{"proposal"}) \geq NumberOfNodes - NumberOfFaultyNodes$
    $\land$ IF $CheckPhase2a(node) \land CheckPhase2b(node)$
        THEN $DecidePhase2a(node) \land EstimatePhase2b(node)$
        ELSE IF $\neg CheckPhase2a(node) \land CheckPhase2b(node)$
              THEN $EstimatePhase2b(node) \land$ UNCHANGED $\langle decisions \rangle$
              ELSE IF $\neg CheckPhase2a(node) \land \neg CheckPhase2b(node)$
                    THEN $EstimatePhase2c(node) \land$ UNCHANGED $\langle decisions \rangle$
                    ELSE FALSE    No other cases possible.
    $\land\ rounds' = [rounds\ \text{EXCEPT}\ ![node] = @ + 1]$   Node goes to the next round.
    $\land$ UNCHANGED $\langle msgs \rangle$
```

Consensus property for *TLC* to check.

```
$ConsensusReached \triangleq$
    $\land\ \forall\, node \in NODES : decisions[node] \in DecisionValues$                All decision values must be "0" or "1".
    $\land\ \forall\, node1, node2 \in NODES : decisions[node1] = decisions[node2]$   All nodes must decide on the same value.
```

The last state of the algorithm when consensus is reached.

```
$LastState \triangleq$
    $ConsensusReached$

$TypeOK \triangleq$
    $\land\ \ \forall\, node \in NODES : rounds[node] \in NodeRounds$
    $\land\ \ \forall\, node \in NODES : estimates[node] \in NodeEstimateValues$
    $\land\ \ \forall\, node \in NODES : decisions[node] \in NodeDecisionValues$
    $\land\ \ msgs \subseteq MessageRecordSet$
```

```
If CHECK_ALL_INITIAL_VALUES is set to TRUE, TLC will check all possible estimate value combinations. Otherwise, initial estimates
  are defined by configuring what nodes estimate a "0" or a "1" value with the sets ESTIMATE_ZERO and ESTIMATE_ONE.
Initial decisions are set to "-1" at the start to ensure ConsensusReached is not true in the initial state.
There are no sent messages in the initial state.
Init ≜
    ∧ rounds = [node ∈ NODES ↦ 1]
    ∧ IF CHECK_ALL_INITIAL_VALUES
        THEN ∃ estimateFunction ∈ NodeEstimateFunctions : estimates = estimateFunction
        ELSE  estimates = [node ∈ NODES ↦ IF node ∈ ESTIMATE_ZERO THEN "0" ELSE "1"]
    ∧ decisions = [node ∈ NODES ↦ "-1"]
    ∧ msgs = {}

Next ≜
    ∃ node ∈ NODES :
        ∧ ¬LastState  Stops TLC when used with deadlock check enabled.
        ∧ ∨ Phase1(node)
          ∨ Phase2(node)

Spec ≜
    ∧ Init
    ∧ □[Next]_vars
    ∧ WF_vars(Next)

ConsensusProperty ≜ ◇□[ConsensusReached]_vars  Eventually, consensus will always be reached.
├──────────────────────────────────────────────────────────────────────────────────────────┤
THEOREM Spec ⇒ □TypeOK  Checked by TLC.
PROOF OMITTED
├──────────────────────────────────────────────────────────────────────────────────────────┤
THEOREM Spec ⇒ ConsensusProperty  Checked by TLC.
PROOF OMITTED
└──────────────────────────────────────────────────────────────────────────────────────────┘
```

Figure 23. The BenOrMsgs TLA$^+$ specification

```
MODULE BenOrAbstSync
```

This is an abstract specification of the *BenOr* consensus algorithm (not the *Byzantine* version). The specification omits message sending entirely by allowing nodes to see each other's estimate
values directly.
Round numbers are also omitted from the specification. Nodes can only go to the next phase of a round when all nodes have completed the previous phase. This means that there is no need to differentiate estimate values with round numbers as was done in the detailed *BenOr* consensus algorithm TLA+ spec.
With less states, it's possible to check for temporal properties using *TLC*, and to more easily investigate how random estimate values can be picked to guarantee consensus.
However, a sequential version of a distributed algorithm doesn't allow for nodes to be in different phases or rounds. This makes *BenOrAbstSync* limited in terms of verifying temporal properties as too many important behaviors are omitted.

```
EXTENDS TLC, Integers, FiniteSets
```

Use the *BenOr.cfg* file.

```
CONSTANTS NODES, FAULTY_NODES, ESTIMATE_ZERO, ESTIMATE_ONE, CHECK_ALL_INITIAL_VALUES

VARIABLE estimates                 Function that returns a given node's current estimated value.
VARIABLE decisions                 Function that returns a given node's current decision value.
VARIABLE proposalMessageValue      Function that returns a proposal message value for a given node.
VARIABLE phaseFlags                Function that returns a value that denotes what phase of a round the node has completed.
VARIABLE currentPhase              A variable that returns the current phase of the round for all nodes.
vars ≜ ⟨estimates, decisions, proposalMessageValue, phaseFlags, currentPhase⟩
```

F must be greater than 0, otherwise the *N* - *F* check for proposal messages won't work.

```
ASSUME ∧ (Cardinality(NODES) ≥ 2 ∗ Cardinality(FAULTY_NODES) + 1)
       ∧ (Cardinality(FAULTY_NODES) ≥ 1)
```

Nodes and their decisions must be specified.

```
ASSUME ∧ NODES ≠ {}
       ∧ ESTIMATE_ONE ≠ {}
       ∧ ESTIMATE_ZERO ≠ {}
```

Decision values must be specified for all nodes.

```
ASSUME Cardinality(ESTIMATE_ONE) + Cardinality(ESTIMATE_ZERO) = Cardinality(NODES)
```

Numbers of nodes and faulty nodes.

```
NumberOfNodes ≜ Cardinality(NODES)
NumberOfFaultyNodes ≜ Cardinality(FAULTY_NODES)
```

Decision values of binary consensus.

```
DecisionValues ≜ {"0", "1"}
```

Set of all possible node decision values including the "-1" used for the initial state.

```
NodeDecisionValues ≜ DecisionValues ∪ {"-1"}
```

Set of all possible proposal message values including the "-1" used to indicate a proposal was not yet sent.

```
ProposalMessageValues ≜ {"?", "0", "1"} ∪ {"-1"}
```

Set of all functions that map nodes to their estimate values.

```
NodeEstimateFunctions ≜ [NODES → DecisionValues]
```

"0" is when the node is able to enter the next round. "1" is when the node finishes the first phase and can move on to the second phase.
"2" is when the node finishes the current round and wants to enter the next round.

```
RoundPhases ≜ {"0", "1", "2"}
```

"-1" is used for the final state of the system.

```
CurrentRoundPhaseValues ≜ RoundPhases ∪ {"-1"}
```

Set of all possible node estimate values. The "-1" is not necessary since all nodes estimate a value of "0" or "1" in the initial state.

```
NodeEstimateValues ≜ DecisionValues
```

Used to print variable values. Must be used with ∧ and in states which are reached by *TLC*.

```
PrintVal(message, expression) ≜ Print(⟨message, expression⟩, TRUE)
```

A set of all nodes other than the specified node.

```
AllOtherNodes(node) ≜
    NODES \ {node}
```

The number of node estimates of a given estimated value made by nodes other than the checking node.

```
HowManyEstimates(checkingNode, estimatedValue) ≜
    Cardinality({node ∈ NODES :
                        ∧ node ≠ checkingNode
                        ∧ estimates[node] = estimatedValue})
```

Sends a proposal message with a value that is in more than N / 2 estimates.

```
SendProposalMessage(node) ≜
    ∃ estimateValue ∈ DecisionValues :
        ∧ HowManyEstimates(node, estimateValue) ∗ 2 > NumberOfNodes
        ∧ proposalMessageValue′ = [proposalMessageValue EXCEPT ![node] = estimateValue]
```

Check if there are more than N / 2 estimates of the same value.

```
CheckPhase1a(node) ≜
    ∃ estimateValue ∈ DecisionValues :
        HowManyEstimates(node, estimateValue) ∗ 2 > NumberOfNodes
```

The first phase of the algorithm. All nodes check whether there are more than N / 2 estimates with the same value v. If such a value exists, then v is sent to all other nodes in a proposal message. v is not necessarily the same as the node's current estimate value. A node can send a value different from its own estimate value to other nodes in a proposal message.
If such a value doesn't exist, the node sends a proposal message with a "?" value.

```
Phase1(node) ≜
    ∧ node ∈ NODES
    ∧ currentPhase = "0"
    ∧ phaseFlags[node] = "0"
    ∧ IF CheckPhase1a(node)
        THEN SendProposalMessage(node)
        ELSE proposalMessageValue′ = [proposalMessageValue EXCEPT ![node] = "?"]
    ∧ phaseFlags′ = [phaseFlags EXCEPT ![node] = "1"]   First phase has been completed by the node.
    ∧ UNCHANGED ⟨estimates, decisions, currentPhase⟩
```

If all nodes have finished the first phase, the second phase of the round can begin.

```
StartPhase2 ≜
    ∧ currentPhase = "0"
    ∧ IF ∀ node ∈ NODES : phaseFlags[node] = "1"
        THEN ∧ currentPhase′ = "1"
             ∧ UNCHANGED ⟨estimates, decisions, proposalMessageValue, phaseFlags⟩
        ELSE UNCHANGED ⟨estimates, decisions, proposalMessageValue, phaseFlags, currentPhase⟩
```

The number of node proposals of a given proposal value made by nodes other than the checking node.

```
HowManyProposals(checkingNode, proposalValue) ≜
    Cardinality({node ∈ NODES :
                        ∧ node ≠ checkingNode
                        ∧ proposalMessageValue[node] = proposalValue})
```

Returns a set of proposal values "0" or "1" that are proposed by at least $F + 1$ nodes.

```
AtLeastNF1Proposals(node) ≜
    {estimateValue ∈ DecisionValues :
        HowManyProposals(node, estimateValue) ≥ NumberOfFaultyNodes + 1}
```

Check for $F + 1$ or more proposal messages with the same value.

```
CheckPhase2a(node) ≜
    AtLeastNF1Proposals(node) ≠ {}
```

Returns a set of proposal values "0" or "1" that have been proposed by at least one node.

```
AtLeast1Proposal(node) ≜
    {estimateValue ∈ DecisionValues :
        HowManyProposals(node, estimateValue) ≥ 1}
```

Check for at least one proposal of a node without the "?" value.

```
CheckPhase2b(node) ≜
    AtLeast1Proposal(node) ≠ {}
```

Decide an estimate value that has been proposed by at least $F + 1$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
DecidePhase2a(node) ≜
    ∃ estimateValue ∈ AtLeastNF1Proposals(node) :
        decisions' = [decisions EXCEPT ![node] = estimateValue]
```

Estimate a proposal value that has been proposed by at least one node. If there are multiple such values, *TLC* will check all possible choices separately.

```
EstimatePhase2b(node) ≜
    ∃ estimateValue ∈ AtLeast1Proposal(node) :
        estimates' = [estimates EXCEPT ![node] = estimateValue]
```

Pick a random estimate value from "0" or "1" for a node. *TLC* will check all possible choices separately.

```
EstimatePhase2c(node) ≜
    ∃ newEstimate ∈ NodeEstimateValues :
        ∧ estimates' = [estimates EXCEPT ![node] = newEstimate]
```

Second phase of the *BenOr* consensus algorithm. All nodes check whether there has been at least $F + 1$ proposal messages with the same value "0" or "1". If there were that many messages, each node updates their decision and estimate values to match the value in the proposal messages.
Otherwise, if there is at least 1 received proposal message with a value "0" or "1", the value is set as the new estimate for the receiving node.
In all other cases, nodes estimate a random value of "0" or "1" for the next round.

```
Phase2(node) ≜
    ∧ node ∈ NODES
    ∧ currentPhase = "1"
    ∧ phaseFlags[node] = "1"
    ∧ IF CheckPhase2a(node) ∧ CheckPhase2b(node)
        THEN DecidePhase2a(node) ∧ EstimatePhase2b(node)   If there are at least F + 1 valid proposals.
        ELSE IF ¬CheckPhase2a(node) ∧ CheckPhase2b(node)
                THEN EstimatePhase2b(node) ∧ UNCHANGED ⟨decisions⟩   If there is at least 1 valid proposal.
                ELSE IF ¬CheckPhase2a(node) ∧ ¬CheckPhase2b(node)
                        THEN EstimatePhase2c(node) ∧ UNCHANGED ⟨decisions⟩   Estimates are randomized.
                        ELSE FALSE   No other cases possible.
    ∧ phaseFlags' = [phaseFlags EXCEPT ![node] = "2"]
    ∧ UNCHANGED ⟨proposalMessageValue, currentPhase⟩
```

If all nodes have finished the second phase, the next round can begin. Sets all node phase and proposal message values to "0" and "-1" accordingly.

```
NextRound ≜
    ∧ currentPhase = "1"
    ∧ IF ∀ node ∈ NODES : phaseFlags[node] = "2"
        THEN ∧ currentPhase' = "0"
             ∧ phaseFlags' = [node ∈ NODES ↦ "0"]
             ∧ proposalMessageValue' = [node ∈ NODES ↦ "-1"]
             ∧ UNCHANGED ⟨estimates, decisions⟩
        ELSE UNCHANGED ⟨estimates, decisions, proposalMessageValue, phaseFlags, currentPhase⟩
```

Consensus property for *TLC* to check.

```
ConsensusReached ≜
    ∧ ∀ node ∈ NODES : decisions[node] ∈ DecisionValues                  All decision values must be "0" or "1".
    ∧ ∀ node1, node2 ∈ NODES : decisions[node1] = decisions[node2]     All nodes must decide on the same value.
```

The last state of the algorithm when consensus is reached and the current phase is set to "-1".

```
LastState ≜
    ∧ ConsensusReached
    ∧ currentPhase ≠ "-1"
    ∧ currentPhase' = "-1"   Set the round phase to "-1" to stop the algorithm.
    ∧ UNCHANGED ⟨estimates, decisions, proposalMessageValue, phaseFlags⟩

TypeOK ≜
    ∧   ∀ node ∈ NODES : estimates[node] ∈ NodeEstimateValues
    ∧   ∀ node ∈ NODES : decisions[node] ∈ NodeDecisionValues
    ∧   ∀ node ∈ NODES : proposalMessageValue[node] ∈ ProposalMessageValues
    ∧   ∀ node ∈ NODES : phaseFlags[node] ∈ RoundPhases
    ∧   currentPhase ∈ CurrentRoundPhaseValues
```

If *CHECK_ALL_INITIAL_VALUES* is set to TRUE, *TLC* will check all possible estimate value combinations. Otherwise, initial estimates are defined by configuring what nodes estimate a "0" or a "1" value with the sets *ESTIMATE_ZERO* and *ESTIMATE_ONE*.
Initial decisions are set to "-1" at the start to ensure *ConsensusReached* is not true in the initial state.

```
Init ≜
    ∧ IF CHECK_ALL_INITIAL_VALUES
        THEN ∃ estimateFunction ∈ NodeEstimateFunctions : estimates = estimateFunction
        ELSE  estimates = [node ∈ NODES ↦ IF node ∈ ESTIMATE_ZERO THEN "0" ELSE  "1"]
    ∧ decisions = [node ∈ NODES ↦ "-1"]
    ∧ proposalMessageValue = [node ∈ NODES ↦ "-1"]
    ∧ phaseFlags = [node ∈ NODES ↦ "0"]
    ∧ currentPhase = "0"

Next ≜
    ∨ ∃ node ∈ NODES :
        ∨ Phase1(node)
        ∨ Phase2(node)
    ∨ StartPhase2
    ∨ NextRound
    ∨ LastState  Stops TLC when used with deadlock check enabled.

Spec ≜
    ∧ Init
    ∧ □[Next]_vars
    ∧ ∀ node ∈ NODES : WF_vars(Phase1(node) ∨ Phase2(node))
    ∧ WF_vars(StartPhase2 ∨ NextRound ∨ LastState)

ConsensusProperty ≜ ◇□[ConsensusReached]_vars  Eventually, consensus will always be reached.
```

```
THEOREM Spec ⇒ □TypeOK  Checked by TLC.
PROOF OMITTED
```

```
THEOREM Spec ⇒ ConsensusProperty  Checked by TLC.
PROOF OMITTED
```

Figure 24. The BenOrAbstSync TLA$^+$ specification

MODULE *BenOrMsgsByz*

This is a specification of the *BenOr Byzantine* consensus algorithm. It is similar to *BenOrMsgs* but with some changes :
- The addition of faulty nodes that always send proposals with the "?" value.
- Changes to the required number of nodes (N) and faulty nodes (F) to comply with the requirements of the *Byzantine* version of the algorithm.
- Addition of conditions for reaching *Byzantine* consesus defined in *ConsensusReachedByzantine*.

Compared to the abstract specification *BenOrAbstByz*, this is a more detailed spec with a lot bigger state space size. This makes checking temporal and safety properties with *TLC* more time consuming. The advantage of this specification is that the state trace generated by *TLC* is easier to examine when trying to understand how round phases progressed since node messages are very descriptive. There are also less variables in this specification.

EXTENDS TLC, $Integers$, $FiniteSets$

Use the *BenOrByz.cfg* file.

CONSTANTS $NODES$, $FAULTY\_NODES$, $ESTIMATE\_ZERO$, $ESTIMATE\_ONE$, $CHECK\_ALL\_INITIAL\_VALUES$

VARIABLE *rounds* Function that returns a given node's current round.
VARIABLE *estimates* Function that returns a given node's current estimated value.
VARIABLE *decisions* Function that returns a given node's current decision value.
VARIABLE *msgs* Set of all sent messages.

$vars \triangleq \langle rounds, estimates, decisions, msgs \rangle$

For the *BenOr Byzantine* consensus protocol version, $N > 5 * F$ must be true. Also, F must be greater than 0, otherwise the $N - F$ check for proposal messages won't work.

ASSUME $\land (Cardinality(NODES) > 5 * Cardinality(FAULTY\_NODES))$
$\quad\quad\quad \land (Cardinality(FAULTY\_NODES) \geq 1)$

Nodes and their decisions must be specified.

ASSUME $\land NODES \neq \{\}$
$\quad\quad\quad \land ESTIMATE\_ONE \neq \{\}$
$\quad\quad\quad \land ESTIMATE\_ZERO \neq \{\}$

Decision values must be specified for all nodes.

ASSUME $Cardinality(ESTIMATE\_ZERO) + Cardinality(ESTIMATE\_ONE) = Cardinality(NODES)$

Numbers of nodes and faulty nodes.

$NumberOfNodes \triangleq Cardinality(NODES)$
$NumberOfFaultyNodes \triangleq Cardinality(FAULTY\_NODES)$

Set of node round numbers.

$NodeRounds \triangleq Nat$

Decision values of binary consensus.

$DecisionValues \triangleq \{\text{"0"}, \text{"1"}\}$

Set of all possible node decision values including the "-1" used for the initial state.

$NodeDecisionValues \triangleq DecisionValues \cup \{\text{"-1"}\}$

Set of all functions that map nodes to their estimate values.

$NodeEstimateFunctions \triangleq [NODES \rightarrow DecisionValues]$

Set of all possible node estimate values. The "-1" is not necessary since all nodes estimate a value of "0" or "1" in the initial state.

$NodeEstimateValues \triangleq DecisionValues$

Possible values and types of messages in the *BenOr* consensus algorithm.

$MessageType \triangleq \{\text{"report"}, \text{"proposal"}\}$
$MessageValues \triangleq \{\text{"0"}, \text{"1"}, \text{"?"}\}$

Each message record also has a round number.

$MessageRecordSet \triangleq [Sender : NODES, Round : Nat, Type : MessageType, Value : MessageValues]$

Used to print variable values. Must be used with $\land$ and in states which are reached by *TLC*.

$PrintVal(message, expression) \triangleq Print(\langle message, expression \rangle, \text{TRUE})$

A set of all nodes other than the specified node.

$AllOtherNodes(node) \triangleq$
$\quad NODES \setminus \{node\}$

Returns TRUE if a node sent the specified message during the specified round.

```
DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue) ≜
    LET messageRecord ≜ [Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ messageValue]
    IN  messageRecord ∈ msgs
```

Returns TRUE if a node sent a message of the specified type with any value ("0", "1", or "?") during the specified round.

```
DidNodeSendMessageOfType(sendingNode, nodeRound, messageType) ≜
    ∃ msgValue ∈ MessageValues :
        LET messageRecord ≜ [Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ msgValue]
        IN  messageRecord ∈ msgs
```

Adds a specific message to the set *msgs* with the sending node and the node's round. If the a message with the same value was already sent during the specified round, it is not added to
the *msgs* set.

```
SendBroadcastMessage(sendingNode, nodeRound, messageType, messageValue) ≜
    ∧ ¬DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue)
    ∧ msgs' = msgs ∪ {[Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ messageValue]}
```

Returns the number of messages with the specified value that were sent by nodes other than *sendingNode*.

```
HowManyMessages(sendingNode, nodeRound, messageType, messageValue) ≜
    Cardinality({msg ∈ msgs :
                    ∧ msg.Sender ≠ sendingNode
                    ∧ msg.Round = nodeRound
                    ∧ msg.Type = messageType
                    ∧ msg.Value = messageValue})
```

Returns the number of messages sent by nodes other than *checkingNode* of the specified message type.

```
HowManyMessagesOfType(checkingNode, nodeRound, messageType) ≜
    Cardinality({node ∈ AllOtherNodes(checkingNode) :
        DidNodeSendMessageOfType(node, nodeRound, messageType)})
```

Returns a set of estimate values that are present in more than $(N + F)/\ 2$ report messages. Only one same value can be present in more than $(N + F)\ /\ 2$ messages.

```
MajorityOfReports(node, nodeRound) ≜
    {estimateValue ∈ NodeEstimateValues :
        HowManyMessages(node, nodeRound, "report", estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes}
```

Check if there are more than $N\ /\ 2$ report messages with the same value.

```
CheckPhase1(node, nodeRound) ≜
    MajorityOfReports(node, nodeRound) ≠ {}
```

After each node sends a report message during the first phase, it waits for $N - F$ report messages and then counts how many messages contain the same estimate value v.
If there are more than $(N + F)/\ 2$ of such messages, the node sends a proposal message with the value that was in more than $(N + F)/\ 2$ report messages.
Otherwise, the node sends a proposal message with the "?" value. All faulty nodes always send proposal messages with the "?" value.

```
WaitForReports(node) ≜
    ∧ node ∈ NODES
    ∧ DidNodeSendMessage(node, rounds[node], "report", estimates[node])
    ∧ HowManyMessagesOfType(node, rounds[node], "report") ≥ NumberOfNodes − NumberOfFaultyNodes
    ∧ IF  ∧ CheckPhase1(node, rounds[node])
          ∧ node ∉ FAULTY_NODES
       THEN ∃ majorityValue ∈ MajorityOfReports(node, rounds[node]) :
                SendBroadcastMessage(node, rounds[node], "proposal", majorityValue)
       ELSE SendBroadcastMessage(node, rounds[node], "proposal", "?")
    ∧ UNCHANGED ⟨rounds, estimates, decisions⟩
```

The first phase of the *BenOr* consensus algorithm. All nodes send report messages with their estimate value v to all other nodes. Each node then checks what kind of values they have received from other nodes. The value v that is sent as a proposal message if it is in the majority of report messages
is not necessarily the same as the node's current estimate value. A node can send a value different from its own estimate value to other nodes in a proposal message.

```
Phase1(node) ≜
    ∧ node ∈ NODES
    ∧ ∨ SendBroadcastMessage(node, rounds[node], "report", estimates[node])
      ∨ WaitForReports(node)
```

```
        ∧ UNCHANGED ⟨rounds, estimates, decisions⟩
```

Returns a set of proposal values "0" or "1" that are proposed by more than $(N + F) / 2$ nodes.

```
MajorityOfProposals(node) ≜
    {estimateValue ∈ NodeEstimateValues :
        HowManyMessages(node, rounds[node], "proposal", estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes}
```

Returns a set of proposal values "0" or "1" that are proposed by at least $F + 1$ nodes.

```
AtLeastNF1Proposals(node) ≜
    {estimateValue ∈ NodeEstimateValues :
        HowManyMessages(node, rounds[node], "proposal", estimateValue) ≥ NumberOfFaultyNodes + 1}
```

Check for more than $(N + F) / 2$ proposal messages with the same value.

```
CheckPhase2a(node) ≜
    MajorityOfProposals(node) ≠ {}
```

Check for $F + 1$ or more proposal messages with the same value.

```
CheckPhase2b(node) ≜
    AtLeastNF1Proposals(node) ≠ {}
```

Decide an estimate value that has been proposed by more than $(N + F) / 2$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
DecidePhase2a(node) ≜
    ∃ decisionValue ∈ MajorityOfProposals(node) :
        decisions' = [decisions EXCEPT ![node] = decisionValue]
```

Estimate a proposal value that has been proposed by at least $F + 1$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
EstimatePhase2b(node) ≜
    ∃ estimateValue ∈ AtLeastNF1Proposals(node) :
        estimates' = [estimates EXCEPT ![node] = estimateValue]
```

Pick a random estimate value from "0" or "1" for a node. *TLC* will check all possible choices separately.

```
EstimatePhase2c(node) ≜
    ∃ randomEstimate ∈ NodeEstimateValues :
        ∧ estimates'    = [estimates EXCEPT ![node] = randomEstimate]
```

The second phase of the *BenOr* consensus algorithm. All nodes that have sent a proposal message wait for $N - F$ of such messages from other nodes. If the node receives more than $(N + F) / 2$ proposal messages with the same value v ("0" or "1"), then
the node sets its decision value to v. Furthermore, if the node receives at least least $F + 1$ proposal messages with the value v, it sets its estimate value to v. Otherwise, the node changes its estimate value to "0" or "1" randomly.

```
Phase2(node) ≜
    ∧ node ∈ NODES
    ∧ DidNodeSendMessageOfType(node, rounds[node], "proposal")
    ∧ HowManyMessagesOfType(node, rounds[node], "proposal") ≥ NumberOfNodes − NumberOfFaultyNodes
    ∧ IF CheckPhase2a(node) ∧ CheckPhase2b(node)
        THEN DecidePhase2a(node) ∧ EstimatePhase2b(node)
        ELSE IF ¬CheckPhase2a(node) ∧ CheckPhase2b(node)
              THEN EstimatePhase2b(node) ∧ UNCHANGED ⟨decisions⟩
              ELSE IF ¬CheckPhase2a(node) ∧ ¬CheckPhase2b(node)
                    THEN EstimatePhase2c(node) ∧ UNCHANGED ⟨decisions⟩
                    ELSE FALSE  No other cases possible.
    ∧ rounds' = [rounds EXCEPT ![node] = @ + 1]  Node goes to the next round.
    ∧ UNCHANGED ⟨msgs⟩
```

Consensus property for *TLC* to check. Consensus is guaranteed only between non-faulty nodes. All decision values must be "0" or "1" and all non-faulty nodes must decide
on the same value.

```
ConsensusReachedByzantine ≜
    ∧ ∀ node ∈ NODES \ FAULTY_NODES : decisions[node] ∈ DecisionValues
    ∧ ∀ node1, node2 ∈ NODES \ FAULTY_NODES : decisions[node1] = decisions[node2]
```

The last state of the algorithm when consensus is reached.

```
LastState ≜
    ConsensusReachedByzantine
```

```
TypeOK ≜
    ∧  ∀ node ∈ NODES : rounds[node] ∈ NodeRounds
    ∧  ∀ node ∈ NODES : estimates[node] ∈ NodeEstimateValues
    ∧  ∀ node ∈ NODES : decisions[node] ∈ NodeDecisionValues
    ∧  msgs ⊆ MessageRecordSet

  If CHECK_ALL_INITIAL_VALUES is set to TRUE, TLC will check all possible estimate value combinations. Otherwise, initial estimates
   are defined by configuring what nodes estimate a "0" or a "1" value with the sets ESTIMATE_ZERO and ESTIMATE_ONE.
  Initial decisions are set to "-1" at the start to ensure ConsensusReached is not true in the initial state.
  There are no sent messages in the initial state.
Init ≜
    ∧ rounds = [node ∈ NODES ↦ 1]
    ∧ IF CHECK_ALL_INITIAL_VALUES
        THEN ∃ estimateFunction ∈ NodeEstimateFunctions : estimates = estimateFunction
        ELSE  estimates = [node ∈ NODES ↦ IF node ∈ ESTIMATE_ZERO THEN "0" ELSE  "1"]
    ∧ decisions = [node ∈ NODES ↦ "-1"]
    ∧ msgs = {}

Next ≜
    ∃ node ∈ NODES :
        ∧ ¬LastState  Stops TLC when used with deadlock check enabled.
        ∧ ∨ Phase1(node)
          ∨ Phase2(node)

Spec ≜
    ∧ Init
    ∧ □[Next]_vars
    ∧ WF_vars(Next)

ConsensusProperty ≜ ◇□[ConsensusReachedByzantine]_vars  Eventually, consensus will always be reached.
├──────────────────────────────────────────────────────────────────────────────┤
THEOREM Spec ⇒ □TypeOK  Checked by TLC.
PROOF OMITTED
├──────────────────────────────────────────────────────────────────────────────┤
THEOREM Spec ⇒ ConsensusProperty  Checked by TLC.
PROOF OMITTED
└──────────────────────────────────────────────────────────────────────────────┘
```

Figure 25. The BenOrMsgsByz TLA^{+} specification

```
MODULE BenOrAbstSyncByz
```

This specification is similar to *BenOrAbstSync* with a few notable changes:
- The addition of faulty nodes that always send proposals with the "?" value.
- Changes to the required number of nodes (N) and faulty nodes (F) to comply with the requirements of the *Byzantine* version of the algorithm.
- Addition of conditions for reaching *Byzantine* consesus defined in *ConsensusReachedByzantine*.

```
EXTENDS TLC, Integers, FiniteSets
```

Use the *BenOrByz.cfg* file.

```
CONSTANTS NODES, FAULTY_NODES, ESTIMATE_ZERO, ESTIMATE_ONE, CHECK_ALL_INITIAL_VALUES

VARIABLE estimates              Function that returns a given node's current estimated value.
VARIABLE decisions              Function that returns a given node's current decision value.
VARIABLE proposalMessageValue   Function that returns a proposal message value for a given node.
VARIABLE phaseFlags             Function that returns a value that denotes what phase of a round the node has completed.
VARIABLE currentPhase           A variable that returns the current phase of the round for all nodes.
vars ≜ ⟨estimates, decisions, proposalMessageValue, phaseFlags, currentPhase⟩
```

For the *BenOr Byzantine* consensus protocol version, $N > 5 * F$ must be true. Also, F must be greater than 0, otherwise the N - F check for proposal messages won't work.

```
ASSUME ∧ (Cardinality(NODES) > 5 ∗ Cardinality(FAULTY_NODES))
       ∧ (Cardinality(FAULTY_NODES) ≥ 1)
```

Nodes and their decisions must be specified.

```
ASSUME ∧ NODES ≠ {}
       ∧ ESTIMATE_ONE ≠ {}
       ∧ ESTIMATE_ZERO ≠ {}
```

Decision values must be specified for all nodes.

```
ASSUME Cardinality(ESTIMATE_ONE) + Cardinality(ESTIMATE_ZERO) = Cardinality(NODES)
```

Numbers of nodes and faulty nodes.

```
NumberOfNodes ≜ Cardinality(NODES)
NumberOfFaultyNodes ≜ Cardinality(FAULTY_NODES)
```

Decision values of binary consensus.

```
DecisionValues ≜ {"0", "1"}
```

Set of all possible node decision values including the "-1" used for the initial state.

```
NodeDecisionValues ≜ DecisionValues ∪ {"-1"}
```

Set of all possible proposal message values including the "-1" used to indicate a proposal was not yet sent.

```
ProposalMessageValues ≜ {"?", "0", "1"} ∪ {"-1"}
```

Set of all functions that map nodes to their estimate values.

```
NodeEstimateFunctions ≜ [NODES → DecisionValues]
```

"0" is when the node is able to enter the next round. "1" is when the node finishes the first phase and can move on to the second phase. "2" is when the node finishes the current round and wants to enter the next round.

```
RoundPhases ≜ {"0", "1", "2"}
```

"-1" is used for the final state of the system.

```
CurrentRoundPhaseValues ≜ RoundPhases ∪ {"-1"}
```

Set of all possible node estimate values. The "-1" is not necessary since all nodes estimate a value of "0" or "1" in the initial state.

```
NodeEstimateValues ≜ DecisionValues
```

Used to print variable values. Must be used with ∧ and in states which are reached by *TLC*.

```
PrintVal(message, expression) ≜ Print(⟨message, expression⟩, TRUE)
```

A set of all nodes other than the specified node.

```
AllOtherNodes(node) ≜
    NODES \ {node}
```

The number of node estimates of a given estimated value made by nodes other than the checking node.

```
HowManyEstimates(checkingNode, estimatedValue) ≜
    Cardinality({node ∈ NODES :
```

```
                    ∧ $node \neq checkingNode$
                    ∧ $estimates[node] = estimatedValue\})$
```

Sends a proposal message with a value that is in more than $(N + F)$ / 2 estimates.

```
$SendProposalMessage(node) \triangleq$
   $\exists\, estimateValue \in DecisionValues$ :
      ∧ $HowManyEstimates(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes$
      ∧ $proposalMessageValue' = [proposalMessageValue$ EXCEPT $![node] = estimateValue]$
```

Check if there are more than $(N + F)$ / 2 estimates of the same value.

```
$CheckPhase1a(node) \triangleq$
   $\exists\, estimateValue \in DecisionValues$ :
      $HowManyEstimates(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes$
```

The first phase of the algorithm. All nodes check whether there are more than $(N + F)$ / 2 estimates with the same value v. If such a value exists, then v is sent to all other nodes in a proposal message. v is not necessarily the same as the node's current estimate value. A node can send
a value different from its own estimate value to other nodes in a proposal message.
If such a value doesn't exist, the node sends a proposal message with a "?" value. A faulty node always sends a proposal message with the "?" value.

```
$Phase1(node) \triangleq$
   ∧ $node \in NODES$
   ∧ $currentPhase = $ "0"
   ∧ $phaseFlags[node] = $ "0"
   ∧ IF ∧ $CheckPhase1a(node)$
        ∧ $node \notin FAULTY\_NODES$
     THEN $SendProposalMessage(node)$
     ELSE $proposalMessageValue' = [proposalMessageValue$ EXCEPT $![node] = $ "?"]
   ∧ $phaseFlags' = [phaseFlags$ EXCEPT $![node] = $ "1"]   First phase has been completed by the node.
   ∧ UNCHANGED $\langle estimates, decisions, currentPhase\rangle$
```

If all nodes have finished the first phase, the second phase of the round can begin.

```
$StartPhase2 \triangleq$
   ∧ $currentPhase = $ "0"
   ∧ IF $\forall\, node \in NODES : phaseFlags[node] = $ "1"
     THEN ∧ $currentPhase' = $ "1"
          ∧ UNCHANGED $\langle estimates, decisions, proposalMessageValue, phaseFlags\rangle$
     ELSE UNCHANGED $\langle estimates, decisions, proposalMessageValue, phaseFlags, currentPhase\rangle$
```

The number of node proposals of a given proposal value made by nodes other than the checking node.

```
$HowManyProposals(checkingNode, proposalValue) \triangleq$
   $Cardinality(\{node \in NODES$ :
                    ∧ $node \neq checkingNode$
                    ∧ $proposalMessageValue[node] = proposalValue\})$
```

Returns a set of proposal values "0" or "1" that are proposed by more than $(N + F)$ / 2 nodes.

```
$AtLeastMajorityProposals(node) \triangleq$
   $\{estimateValue \in DecisionValues$ :
      $HowManyProposals(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes\}$
```

Returns a set of proposal values "0" or "1" that are proposed by at least $F + 1$ nodes.

```
$AtLeastNF1Proposals(node) \triangleq$
   $\{estimateValue \in DecisionValues$ :
      $HowManyProposals(node, estimateValue) \geq NumberOfFaultyNodes + 1\}$
```

Check for $F + 1$ or more proposal messages with the same value.

```
$CheckPhase2a(node) \triangleq$
   $AtLeastMajorityProposals(node) \neq \{\}$
```

Check for $F + 1$ or more proposal messages with the same value.

```
$CheckPhase2b(node) \triangleq$
   $AtLeastNF1Proposals(node) \neq \{\}$
```

Decide an estimate value that has been proposed by more than $(N + F)$ / 2 nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$DecidePhase2a(node) \triangleq$
    $\exists\, estimateValue \in AtLeastMajorityProposals(node)$ :
        $decisions' = [decisions$ EXCEPT $![node] = estimateValue]$
```

Estimate a proposal value that has been proposed by at least $F + 1$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$EstimatePhase2b(node) \triangleq$
    $\exists\, estimateValue \in AtLeastNF1Proposals(node)$ :
        $estimates' = [estimates$ EXCEPT $![node] = estimateValue]$
```

Pick a random estimate value from "0" or "1" for a node. *TLC* will check all possible choices separately.

```
$EstimatePhase2c(node) \triangleq$
    $\exists\, newEstimate \in NodeEstimateValues$ :
        $\land\, estimates' = [estimates$ EXCEPT $![node] = newEstimate]$
```

Second phase of the *BenOr* consensus algorithm. All nodes check whether there have been more than $(N + F)$ / 2 proposal messages with the same value "0" or "1". If there were that many messages, each node updates their decision and estimate values to match the
value in the proposal messages.
Otherwise, if there are at least $F + 1$ received proposal messages with the same value value "0" or "1", the value is set as the new estimate
for the receiving node.
In all other cases, nodes estimate a random value of "0" or "1" for the next round.

```
$Phase2(node) \triangleq$
    $\land\, node \in NODES$
    $\land\, currentPhase =$ "1"
    $\land\, phaseFlags[node] =$ "1"
    $\land$ IF $CheckPhase2a(node) \land CheckPhase2b(node)$
        THEN $DecidePhase2a(node) \land EstimatePhase2b(node)$   If there are more than $(N + F)$ / 2 valid proposals.
        ELSE IF $\neg CheckPhase2a(node) \land CheckPhase2b(node)$
                THEN $EstimatePhase2b(node) \land$ UNCHANGED $\langle decisions \rangle$   If there are at least $F + 1$ valid proposal messages.
                ELSE IF $\neg CheckPhase2a(node) \land \neg CheckPhase2b(node)$
                        THEN $EstimatePhase2c(node) \land$ UNCHANGED $\langle decisions \rangle$   Estimates are randomized.
                        ELSE FALSE   No other cases possible.
    $\land\, phaseFlags' = [phaseFlags$ EXCEPT $![node] =$ "2"$]$
    $\land$ UNCHANGED $\langle proposalMessageValue, currentPhase \rangle$
```

If all nodes have finished the second phase, the next round can begin. Sets all node phase and proposal message values to "0" and "-1" accordingly.

```
$NextRound \triangleq$
    $\land\, currentPhase =$ "1"
    $\land$ IF $\forall\, node \in NODES : phaseFlags[node] =$ "2"
        THEN $\land\, currentPhase' =$ "0"
             $\land\, phaseFlags' = [node \in NODES \mapsto$ "0"$]$
             $\land\, proposalMessageValue' = [node \in NODES \mapsto$ "-1"$]$
             $\land$ UNCHANGED $\langle estimates, decisions \rangle$
        ELSE UNCHANGED $\langle estimates, decisions, proposalMessageValue, phaseFlags, currentPhase \rangle$
```

Consensus property for *TLC* to check. Consensus is guaranteed only between non-faulty nodes. All decision values must be "0" or "1" and all non-faulty nodes must decide
on the same value.

```
$ConsensusReachedByzantine \triangleq$
    $\land\, \forall\, node \in NODES \setminus FAULTY\_NODES : decisions[node] \in DecisionValues$
    $\land\, \forall\, node1, node2 \in NODES \setminus FAULTY\_NODES : decisions[node1] = decisions[node2]$
```

The last state of the algorithm when consensus is reached and the current phase is set to "-1".

```
$LastState \triangleq$
    $\land\, ConsensusReachedByzantine$
    $\land\, currentPhase \neq$ "-1"
    $\land\, currentPhase' =$ "-1"   Set the round phase to "-1" to stop the algorithm.
    $\land$ UNCHANGED $\langle estimates, decisions, proposalMessageValue, phaseFlags \rangle$

$TypeOK \triangleq$
    $\land\quad \forall\, node \in NODES : estimates[node] \in NodeEstimateValues$
    $\land\quad \forall\, node \in NODES : decisions[node] \in NodeDecisionValues$
    $\land\quad \forall\, node \in NODES : proposalMessageValue[node] \in ProposalMessageValues$
```

```
$\land$  $\forall node \in NODES : phaseFlags[node] \in RoundPhases$
$\land$  $currentPhase \in CurrentRoundPhaseValues$
```

If *CHECK_ALL_INITIAL_VALUES* is set to TRUE, *TLC* will check all possible estimate value combinations. Otherwise, initial estimates are defined by configuring what nodes estimate a "0" or a "1" value with the sets *ESTIMATE_ZERO* and *ESTIMATE_ONE*.
Initial decisions are set to "-1" at the start to ensure *ConsensusReached* is not true in the initial state.

```
$Init \triangleq$
    $\land$ IF $CHECK\_ALL\_INITIAL\_VALUES$
        THEN $\exists\, estimateFunction \in NodeEstimateFunctions : estimates = estimateFunction$
        ELSE  $estimates = [node \in NODES \mapsto$ IF $node \in ESTIMATE\_ZERO$ THEN "0" ELSE "1"$]$
    $\land$ $decisions = [node \in NODES \mapsto$ "-1"$]$
    $\land$ $proposalMessageValue = [node \in NODES \mapsto$ "-1"$]$
    $\land$ $phaseFlags = [node \in NODES \mapsto$ "0"$]$
    $\land$ $currentPhase =$ "0"

$Next \triangleq$
    $\lor$ $\exists\, node \in NODES :$
        $\lor$ $Phase1(node)$
        $\lor$ $Phase2(node)$
    $\lor$ $StartPhase2$
    $\lor$ $NextRound$
    $\lor$ $LastState$  Stops TLC when used with deadlock check enabled.

$Spec \triangleq$
    $\land$ $Init$
    $\land$ $\Box[Next]_{vars}$
    $\land$ $\forall\, node \in NODES : \mathrm{WF}_{vars}(Phase1(node) \lor Phase2(node))$
    $\land$ $\mathrm{WF}_{vars}(StartPhase2 \lor NextRound \lor LastState)$

$ConsensusProperty \triangleq \Diamond\Box[ConsensusReachedByzantine]_{vars}$  Eventually, consensus will always be reached.

THEOREM $Spec \Rightarrow \Box TypeOK$  Checked by TLC.
PROOF OMITTED

THEOREM $Spec \Rightarrow ConsensusProperty$  Checked by TLC.
PROOF OMITTED
```

Figure 26. The BenOrAbstSyncByz TLA$^+$ specification

MODULE *BenOrAbstByz*

This is an abstract specification of the *BenOr Byzantine* consensus algorithm. The specification omits message sending entirely by allowing nodes to see each other's estimate
values directly.
Changes made compared to *BenOrAbstSync* :
 – The specification now allows for nodes to be in different rounds and phases.
 - Added variables for tracking estimate and proposal value history.
 - Added a finite number of rounds for nodes to be able to be in different rounds.
If there is no message log, nodes have to somehow keep all the estimate and decision values for all possible rounds. If there was a msgs set, this wouldn't be necessary because each node could just look at the messages that were previously sent and added to the msgs set since each message has a round number associated with it.
Why is it necessary to store all estimates and decisions for all node rounds? Because if one node is in the previous round and it has to look up the other node's previous *rounds'* estimates, they won't be accessible since all the other nodes are already in the next round. An opportunity must be given for nodes to "catch up" with other nodes based on the algorithm's communication protocol.
With less states, it's possible to check for temporal properties using *TLC*, and to more easily investigate how random estimate values can be picked to guarantee consensus.

EXTENDS TLC, $Integers$, $FiniteSets$

Use the *BenOrByz.cfg* file.
CONSTANTS $NODES$, $FAULTY\_NODES$, $ESTIMATE\_ZERO$, $ESTIMATE\_ONE$, $CHECK\_ALL\_INITIAL\_VALUES$

VARIABLE $estimatesAtRound$	Function that returns a function of all the node's estimates at the specified round.
VARIABLE $proposalsAtRound$	Function that returns a function of all the node's proposal values at the specified round.
VARIABLE $decisions$	Function that returns a given node's current decision value.
VARIABLE $rounds$	Function that returns a given node's current round.

$vars \triangleq \langle estimatesAtRound, proposalsAtRound, decisions, rounds \rangle$

For the *BenOr Byzantine* consensus protocol version, $N > 5 * F$ must be true. Also, F must be greater than 0, otherwise the $N - F$ check for proposal messages won't work.

ASSUME $\land (Cardinality(NODES) > 5 * Cardinality(FAULTY\_NODES))$
       $\land (Cardinality(FAULTY\_NODES) \geq 1)$

Nodes and their decisions must be specified.

ASSUME $\land NODES \neq \{\}$
       $\land ESTIMATE\_ONE \neq \{\}$
       $\land ESTIMATE\_ZERO \neq \{\}$

Decision values must be specified for all nodes.

ASSUME $Cardinality(ESTIMATE\_ONE) + Cardinality(ESTIMATE\_ZERO) = Cardinality(NODES)$

Numbers of nodes and faulty nodes.
$NumberOfNodes \triangleq Cardinality(NODES)$
$NumberOfFaultyNodes \triangleq Cardinality(FAULTY\_NODES)$

Only a limited number of rounds is modeled.
$NodeRounds \triangleq 1 \,..\, 3$

Decision values of binary consensus.
$DecisionValues \triangleq \{\text{"0"}, \text{"1"}\}$

Set of all possible node decision values including the "-1" used for the initial state.
$NodeDecisionValues \triangleq DecisionValues \cup \{\text{"-1"}\}$

Set of possible proposal values.
$ProposalValues \triangleq \{\text{"?"}, \text{"0"}, \text{"1"}\}$

Set of all possible proposal message values including the "-1" used to indicate a proposal was not yet sent.
$ProposalMessageValues \triangleq ProposalValues \cup \{\text{"-1"}\}$

Set of all possible node estimate values. The "-1" is used to track whether a node has sent a report message or not.
$NodeEstimateValues \triangleq DecisionValues \cup \{\text{"-1"}\}$

Set of all functions that map nodes to their estimate values and "-1".
$AllNodeEstimateFunctions \triangleq [NODES \rightarrow NodeEstimateValues]$

Set of all functions that map nodes to their estimate values "0" or "1".
$ValidNodeEstimateFunctions \triangleq [NODES \rightarrow DecisionValues]$

Set of all functions that map nodes to their proposal message values.

```
NodeProposalFunctions ≜ [NODES → ProposalMessageValues]
```

Used to print variable values. Must be used with ∧ and in states which are reached by *TLC*.

```
PrintVal(message, expression) ≜ Print(⟨message, expression⟩, TRUE)
```

Used to print the values of all variables.

```
PrintAllVariableValues ≜
    ∧ PrintVal("Estimates at round: ", estimatesAtRound)
    ∧ PrintVal("Proposals at round: ", proposalsAtRound)
    ∧ PrintVal("Decisions: ", decisions)
    ∧ PrintVal("Rounds: ", rounds)
```

A set of all nodes other than the specified node.

```
AllOtherNodes(node) ≜
    NODES \ {node}
```

The number of node estimates of a given estimated value made by nodes other than the checking node in the same round.

```
HowManyEstimates(checkingNode, estimatedValue) ≜
    Cardinality({node ∈ NODES :
                        ∧ node ≠ checkingNode
                        ∧ estimatesAtRound[rounds[checkingNode]][node] = estimatedValue})
```

Sends a proposal message with a value that is in more than $(N + F)$ / 2 estimates.

```
SendProposalMessage(node) ≜
    ∃ estimateValue ∈ DecisionValues :
      ∧ HowManyEstimates(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes
      ∧ proposalsAtRound' = [proposalsAtRound EXCEPT ![rounds[node]][node] = estimateValue]
```

Check if there are more than $(N + F)$ / 2 estimates of the same value.

```
CheckPhase1a(node) ≜
    ∃ estimateValue ∈ DecisionValues :
      HowManyEstimates(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes
```

Wait for $N - F$ nodes with estimate values other than "-1" in the same round.

```
WaitForReports(waitingNode) ≜
    Cardinality({node ∈ NODES :
       ∧ node ≠ waitingNode
       ∧ estimatesAtRound[rounds[waitingNode]][node] ∈ DecisionValues}) ≥ NumberOfNodes − NumberOfFaultyNodes
```

The first phase of the algorithm. All nodes check whether there are more than $(N + F)$ / 2 estimates with the same value v. If such a value exists, then v is sent to all other nodes in a proposal message. v is not necessarily the same as the node's current estimate value. A node can send
a value different from its own estimate value to other nodes in a proposal message.
If such a value doesn't exist, the node sends a proposal message with a "?" value. A faulty node always sends a proposal message with the "?" value.

```
Phase1(node) ≜
    ∧ node ∈ NODES
    ∧ proposalsAtRound[rounds[node]][node] ∉ ProposalValues
    ∧ WaitForReports(node)
    ∧ IF  ∧ CheckPhase1a(node)
          ∧ node ∉ FAULTY_NODES
      THEN SendProposalMessage(node)
      ELSE proposalsAtRound' = [proposalsAtRound EXCEPT ![rounds[node]][node] = "?"]
    ∧ UNCHANGED ⟨estimatesAtRound, decisions, rounds⟩
```

The number of node proposals of a given proposal value made by nodes other than the checking node in the same round.

```
HowManyProposals(checkingNode, proposalValue) ≜
    Cardinality({node ∈ NODES :
                        ∧ node ≠ checkingNode
                        ∧ proposalsAtRound[rounds[checkingNode]][node] = proposalValue})
```

Returns a set of proposal values "0" or "1" that are proposed by more than $(N + F)$ / 2 nodes.

```
AtLeastMajorityProposals(node) ≜
    {estimateValue ∈ DecisionValues :
        HowManyProposals(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes}

 Returns a set of proposal values "0" or "1" that are proposed by at least F + 1 nodes.
AtLeastNF1Proposals(node) ≜
    {estimateValue ∈ DecisionValues :
        HowManyProposals(node, estimateValue) ≥ NumberOfFaultyNodes + 1}

 Check for F + 1 or more proposal messages with the same value.
CheckPhase2a(node) ≜
    AtLeastMajorityProposals(node) ≠ {}

 Check for F + 1 or more proposal messages with the same value.
CheckPhase2b(node) ≜
    AtLeastNF1Proposals(node) ≠ {}

 Decide an estimate value that has been proposed by more than (N + F) / 2 nodes. If there are multiple such values, TLC will check all
 possible choices separately.
DecidePhase2a(node) ≜
    ∃ estimateValue ∈ AtLeastMajorityProposals(node) :
      decisions' = [decisions EXCEPT ![node] = estimateValue]

 Estimate a proposal value that has been proposed by at least F + 1 nodes. If there are multiple such values, TLC will check all possible
 choices separately.
EstimatePhase2b(node) ≜
    ∃ estimateValue ∈ AtLeastNF1Proposals(node) :
      estimatesAtRound' = [estimatesAtRound EXCEPT ![rounds[node]][node] = estimateValue]

 Pick a random estimate value from "0" or "1" for a node. TLC will check all possible choices separately.
EstimatePhase2c(node) ≜
    ∃ newEstimate ∈ DecisionValues :
      estimatesAtRound' = [estimatesAtRound EXCEPT ![rounds[node]][node] = newEstimate]

 Wait for N − F proposal messages with values other than "-1" in the same round.
WaitForProposals(waitingNode) ≜
    Cardinality({node ∈ NODES :
        ∧ node ≠ waitingNode
        ∧ proposalsAtRound[rounds[waitingNode]][node] ∈ ProposalValues}) ≥ NumberOfNodes − NumberOfFaultyNodes

 Second phase of the BenOr consensus algorithm. All nodes check whether there have been more than (N + F) / 2 proposal messages with
 the same value "0" or "1". If there were that many messages, each node updates their decision and estimate values to match the
  value in the proposal messages.
 Otherwise, if there are at least F + 1 received proposal messages with the same value value "0" or "1", the value is set as the new estimate
  for the receiving node.
 In all other cases, nodes estimate a random value of "0" or "1" for the next round.
Phase2(node) ≜
    ∧ node ∈ NODES
    ∧ WaitForProposals(node)
    ∧ IF CheckPhase2a(node) ∧ CheckPhase2b(node)
        THEN DecidePhase2a(node) ∧ EstimatePhase2b(node)  If there are more than (N + F) / 2 valid proposals.
        ELSE IF ¬CheckPhase2a(node) ∧ CheckPhase2b(node)
               THEN EstimatePhase2b(node) ∧ UNCHANGED ⟨decisions⟩  If there are at least F + 1 valid proposals.
               ELSE IF ¬CheckPhase2a(node) ∧ ¬CheckPhase2b(node)
                      THEN EstimatePhase2c(node) ∧ UNCHANGED ⟨decisions⟩  Estimates are randomized.
                      ELSE FALSE  No other cases possible.
    ∧ rounds' = [rounds EXCEPT ![node] = @ + 1]
    ∧ UNCHANGED ⟨proposalsAtRound⟩

 Set the node's next round's estimate to the previous round's estimate value so that other nodes are aware that the node has entered the
  next round. In the previous round the estimate will be some value other than "-1".
SetNextRoundEstimate(node) ≜
    ∧ node ∈ NODES
    ∧ estimatesAtRound[rounds[node]][node] = "-1"
    ∧ estimatesAtRound' = [estimatesAtRound EXCEPT ![rounds[node]][node] = estimatesAtRound[rounds[node] − 1][node]]
    ∧ UNCHANGED ⟨proposalsAtRound, decisions, rounds⟩
```

Consensus property for *TLC* to check. Consensus is guaranteed only between non-faulty nodes. All decision values must be "0" or "1" and all non-faulty nodes must decide
on the same value.

```
$ConsensusReachedByzantine \triangleq$
    $\land \forall node \in NODES \setminus FAULTY\_NODES : decisions[node] \in DecisionValues$
    $\land \forall node1, node2 \in NODES \setminus FAULTY\_NODES : decisions[node1] = decisions[node2]$
```

The last state of the algorithm when consensus is reached.

```
$LastState \triangleq$
    $\land ConsensusReachedByzantine$

$TypeOK \triangleq$
    $\land \quad \forall round \in NodeRounds : estimatesAtRound[round] \in AllNodeEstimateFunctions$
    $\land \quad \forall round \in NodeRounds : proposalsAtRound[round] \in NodeProposalFunctions$
    $\land \quad \forall node \in NODES : decisions[node] \in NodeDecisionValues$
    $\land \quad \forall node \in NODES : rounds[node] \in NodeRounds$
```

If *CHECK_ALL_INITIAL_VALUES* is set to TRUE, *TLC* will check all possible estimate value combinations. Otherwise, initial estimates are defined by configuring what nodes estimate a "0" or a "1" value with the sets *ESTIMATE_ZERO* and *ESTIMATE_ONE*.
Estimates for all other rounds are set to "-1". Proposal message values for all rounds are set to "-1". Initial decisions are set to "-1" at the start to ensure *ConsensusReached* is not true
in the initial state.

```
$Init \triangleq$
    $\land$ IF $CHECK\_ALL\_INITIAL\_VALUES$
       THEN $\exists estimateFunction \in ValidNodeEstimateFunctions :$
                $estimatesAtRound = [round \in NodeRounds \mapsto$ IF $round = 1$
                                                                    THEN $estimateFunction$
                                                                    ELSE $[node \in NODES \mapsto$ "-1"$]]$
       ELSE $estimatesAtRound = [round \in NodeRounds \mapsto$ IF $round = 1$
                                                                    THEN $[node \in NODES \mapsto$
                                                                        IF $node \in ESTIMATE\_ZERO$
                                                                         THEN "0"
                                                                         ELSE "1"$]$
                                                                    ELSE $[node \in NODES \mapsto$ "-1"$]]$
    $\land proposalsAtRound = [round \in NodeRounds \mapsto [node \in NODES \mapsto$ "-1"$]]$
    $\land decisions = [node \in NODES \mapsto$ "-1"$]$
    $\land rounds = [node \in NODES \mapsto 1]$

$Next \triangleq$
    $\exists node \in NODES :$
       $\land \neg LastState$   Stops TLC when used with deadlock check enabled.
       $\land \lor Phase1(node)$
          $\lor Phase2(node)$
          $\lor SetNextRoundEstimate(node)$

$Spec \triangleq$
    $\land Init$
    $\land \Box[Next]_{vars}$
    $\land \forall node \in NODES :$
       $\text{WF}_{vars}(Phase1(node) \lor Phase2(node) \lor SetNextRoundEstimate(node))$

$ConsensusProperty \triangleq \Diamond\Box[ConsensusReachedByzantine]_{vars}$   Eventually, consensus will always be reached.
```

THEOREM $Spec \Rightarrow \Box TypeOK$ Checked by *TLC*.
PROOF OMITTED

THEOREM $Spec \Rightarrow ConsensusProperty$ Checked by *TLC*.
PROOF OMITTED

Figure 27. The BenOrAbstByz TLA^{+} specification

MODULE *BenOrAbstByzCC*

This specification is mostly the same as *BenOrAbstByz* with some additional changes :
 – Added a system action for tossing a common coin with a new variable *commonCoin*.
 - Nodes can't estimate values randomly unless the common coin was tossed.
 - All nodes randomly estimate the same value that is equal to the common coin value.
 - A new temporal property *CommonCoinProperty* checks whether the coin is tossed for all rounds.
The common coin is used to model successful termination of the *BenOr Byzantine* consensus algorithm with all possible inputs given a number of nodes N.
With the common coin, *Byzantine* consensus is reached for all possible initial estimate values.

EXTENDS *TLC*, *Integers*, *FiniteSets*

Use the *BenOrByzCC.cfg* file.
CONSTANTS *NODES*, *FAULTY_NODES*, *ESTIMATE_ZERO*, *ESTIMATE_ONE*, *CHECK_ALL_INITIAL_VALUES*

VARIABLE *estimatesAtRound*	Function that returns a function of all the node's estimates at the specified round.
VARIABLE *proposalsAtRound*	Function that returns a function of all the node's proposal values at the specified round.
VARIABLE *decisions*	Function that returns a given node's current decision value.
VARIABLE *rounds*	Function that returns a given node's current round.
VARIABLE *commonCoin*	Models the Common Coin.

$vars \triangleq \langle estimatesAtRound, proposalsAtRound, decisions, rounds, commonCoin \rangle$

For the *BenOr Byzantine* consensus protocol version, $N > 5 * F$ must be true. Also, F must be greater than 0, otherwise the $N - F$ check for proposal messages won't work.

ASSUME $\land (Cardinality(NODES) > 5 * Cardinality(FAULTY\_NODES))$
 $\land (Cardinality(FAULTY\_NODES) \geq 1)$

Nodes and their decisions must be specified.
ASSUME $\land NODES \neq \{\}$
 $\land ESTIMATE\_ONE \neq \{\}$
 $\land ESTIMATE\_ZERO \neq \{\}$

Decision values must be specified for all nodes.
ASSUME $Cardinality(ESTIMATE\_ONE) + Cardinality(ESTIMATE\_ZERO) = Cardinality(NODES)$

Numbers of nodes and faulty nodes.
$NumberOfNodes \triangleq Cardinality(NODES)$
$NumberOfFaultyNodes \triangleq Cardinality(FAULTY\_NODES)$

Only a limited number of rounds is modeled.
$NodeRounds \triangleq 1 \,..\, 3$

Decision values of binary consensus.
$DecisionValues \triangleq \{\text{“0”}, \text{“1”}\}$

Set of all possible node decision values including the “-1” used for the initial state.
$NodeDecisionValues \triangleq DecisionValues \cup \{\text{“-1”}\}$

Set of possible proposal values.
$ProposalValues \triangleq \{\text{“?”}, \text{“0”}, \text{“1”}\}$

Set of all possible proposal message values including the “-1” used to indicate a proposal was not yet sent.
$ProposalMessageValues \triangleq ProposalValues \cup \{\text{“-1”}\}$

Set of all possible node estimate values. The “-1” is used to track whether a node has sent a report message or not.
$NodeEstimateValues \triangleq DecisionValues \cup \{\text{“-1”}\}$

Set of all functions that map nodes to their estimate values and “-1”.
$AllNodeEstimateFunctions \triangleq [NODES \rightarrow NodeEstimateValues]$

Set of all functions that map nodes to their estimate values “0” or “1”.
$ValidNodeEstimateFunctions \triangleq [NODES \rightarrow DecisionValues]$

Set of all functions that map nodes to their proposal message values.
$NodeProposalFunctions \triangleq [NODES \rightarrow ProposalMessageValues]$

Set of all possible Common Coin values. “-1” means that the common coin was not tossed yet for that round.

```
CommonCoinValues ≜ {"-1", "0", "1"}

Used to print variable values. Must be used with ∧ and in states which are reached by TLC.
PrintVal(message, expression) ≜ Print(⟨message, expression⟩, TRUE)

Used to print the values of all variables.
PrintAllVariableValues ≜
    ∧ PrintVal("Estimates at round: ", estimatesAtRound)
    ∧ PrintVal("Proposals at round: ", proposalsAtRound)
    ∧ PrintVal("Decisions: ", decisions)
    ∧ PrintVal("Rounds: ", rounds)
    ∧ PrintVal("Common Coin:", commonCoin)

A set of all nodes other than the specified node.
AllOtherNodes(node) ≜
    NODES \ {node}

The number of node estimates of a given estimated value made by nodes other than the checking node in the same round.
HowManyEstimates(checkingNode, estimatedValue) ≜
    Cardinality({node ∈ NODES :
        ∧ node ≠ checkingNode
        ∧ estimatesAtRound[rounds[checkingNode]][node] = estimatedValue})

Sends a proposal message with a value that is in more than (N + F) / 2 estimates.
SendProposalMessage(node) ≜
    ∃ estimateValue ∈ DecisionValues :
       ∧ HowManyEstimates(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes
       ∧ proposalsAtRound' = [proposalsAtRound EXCEPT ![rounds[node]][node] = estimateValue]

Check if there are more than (N + F) / 2 estimates of the same value.
CheckPhase1a(node) ≜
    ∃ estimateValue ∈ DecisionValues :
       HowManyEstimates(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes

Wait for N − F nodes with estimate values other than "-1" in the same round.
WaitForReports(waitingNode) ≜
    Cardinality({node ∈ NODES :
        ∧ node ≠ waitingNode
        ∧ estimatesAtRound[rounds[waitingNode]][node] ∈ DecisionValues}) ≥ NumberOfNodes − NumberOfFaultyNodes

The first phase of the algorithm. All nodes check whether there are more than (N + F) / 2 estimates with the same value v. If such a value
exists, then v is sent to all other nodes in a proposal message. v is not necessarily the same as the node's current estimate value. A node
can send
 a value different from its own estimate value to other nodes in a proposal message.
If such a value doesn't exist, the node sends a proposal message with a "?" value. A faulty node always sends a proposal message with the
"?" value.
Phase1(node) ≜
    ∧ node ∈ NODES
    ∧ proposalsAtRound[rounds[node]][node] ∉ ProposalValues
    ∧ WaitForReports(node)
    ∧ IF  ∧ CheckPhase1a(node)
          ∧ node ∉ FAULTY_NODES
       THEN SendProposalMessage(node)
       ELSE proposalsAtRound' = [proposalsAtRound EXCEPT ![rounds[node]][node] = "?"]
    ∧ UNCHANGED ⟨estimatesAtRound, decisions, rounds, commonCoin⟩

The number of node proposals of a given proposal value made by nodes other than the checking node in the same round.
HowManyProposals(checkingNode, proposalValue) ≜
    Cardinality({node ∈ NODES :
        ∧ node ≠ checkingNode
        ∧ proposalsAtRound[rounds[checkingNode]][node] = proposalValue})

Returns a set of proposal values "0" or "1" that are proposed by more than (N + F) / 2 nodes.
AtLeastMajorityProposals(node) ≜
    {estimateValue ∈ DecisionValues :
        $HowManyProposals(node,\ estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes\}$
```

Returns a set of proposal values "0" or "1" that are proposed by at least $F + 1$ nodes.

```
$AtLeastNF1Proposals(node) \triangleq$
    $\{estimateValue \in DecisionValues :$
        $HowManyProposals(node,\ estimateValue) \geq NumberOfFaultyNodes + 1\}$
```

Check for $F + 1$ or more proposal messages with the same value.

```
$CheckPhase2a(node) \triangleq$
    $AtLeastMajorityProposals(node) \neq \{\}$
```

Check for $F + 1$ or more proposal messages with the same value.

```
$CheckPhase2b(node) \triangleq$
    $AtLeastNF1Proposals(node) \neq \{\}$
```

Decide an estimate value that has been proposed by more than $(N + F)\ /\ 2$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$DecidePhase2a(node) \triangleq$
    $\exists\, estimateValue \in AtLeastMajorityProposals(node) :$
        $decisions' = [decisions\ \text{EXCEPT}\ ![node] = estimateValue]$
```

Estimate a proposal value that has been proposed by at least $F + 1$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$EstimatePhase2b(node) \triangleq$
    $\exists\, estimateValue \in AtLeastNF1Proposals(node) :$
        $estimatesAtRound' = [estimatesAtRound\ \text{EXCEPT}\ ![rounds[node]][node] = estimateValue]$
```

Pick an estimate value from "0" or "1" for a node based on the common coin of the current round.

```
$EstimatePhase2c(node) \triangleq$
    $\land\ commonCoin[rounds[node]] \neq$ "-1"
    $\land\ estimatesAtRound' = [estimatesAtRound\ \text{EXCEPT}\ ![rounds[node]][node] = commonCoin[rounds[node]]]$
```

Wait for $N - F$ proposal messages with values other than "-1" in the same round.

```
$WaitForProposals(waitingNode) \triangleq$
    $Cardinality(\{node \in NODES :$
        $\land\ node \neq waitingNode$
        $\land\ proposalsAtRound[rounds[waitingNode]][node] \in ProposalValues\}) \geq NumberOfNodes - NumberOfFaultyNodes$
```

Second phase of the *BenOr* consensus algorithm. All nodes check whether there have been more than $(N + F)\ /\ 2$ proposal messages with the same value "0" or "1". If there were that many messages, each node updates their decision and estimate values to match the value in the proposal messages.
Otherwise, if there are at least $F + 1$ received proposal messages with the same value value "0" or "1", the value is set as the new estimate for the receiving node.
In all other cases, nodes estimate a random value of "0" or "1" for the next round.

```
$Phase2(node) \triangleq$
    $\land\ node \in NODES$
    $\land\ WaitForProposals(node)$
    $\land$ IF $CheckPhase2a(node) \land CheckPhase2b(node)$
        THEN $DecidePhase2a(node) \land EstimatePhase2b(node)$    If there are more than $(N + F)\ /\ 2$ valid proposals.
        ELSE IF $\neg CheckPhase2a(node) \land CheckPhase2b(node)$
            THEN $EstimatePhase2b(node) \land$ UNCHANGED $\langle decisions \rangle$    If there are at least $F + 1$ valid proposals.
            ELSE IF $\neg CheckPhase2a(node) \land \neg CheckPhase2b(node)$
                THEN $EstimatePhase2c(node) \land$ UNCHANGED $\langle decisions \rangle$    Estimates are randomized.
                ELSE FALSE    No other cases possible.
    $\land\ rounds' = [rounds\ \text{EXCEPT}\ ![node] = @ + 1]$
    $\land$ UNCHANGED $\langle proposalsAtRound,\ commonCoin \rangle$
```

Set the node's next round's estimate to the previous round's estimate value so that other nodes are aware that the node has entered the next round. In the previous round the estimate will be some valid value other than "-1".

```
$SetNextRoundEstimate(node) \triangleq$
    $\land\ node \in NODES$
    $\land\ estimatesAtRound[rounds[node]][node] =$ "-1"
    $\land\ estimatesAtRound' = [estimatesAtRound\ \text{EXCEPT}\ ![rounds[node]][node] = estimatesAtRound[rounds[node] - 1][node]]$
    $\land$ UNCHANGED $\langle proposalsAtRound,\ decisions,\ rounds,\ commonCoin \rangle$
```

Set the Common Coin value for rounds that have been reached by nodes. If in the round at least 1 node has already estimated some value non-randomly, then that round's common coin
value will be set to the node's estimate value (the Common Coin was lucky).
Otherwise, the Common Coin value is set to "0" or "1" and *TLC* will check behaviors when all nodes randomly estimate "0" or "1" together.

```
TossCommonCoin ≜
    ∧ ∃ round ∈ NodeRounds :
        ∧ commonCoin[round] = "-1"
        ∧ ∨ ∃ node ∈ NODES \ FAULTY_NODES :   If at least 1 node already estimated some value in phase 2b.
                ∧ estimatesAtRound[round][node] ≠ "-1"
                ∧ WaitForProposals(node)
                ∧ ∨ CheckPhase2a(node) ∧ CheckPhase2b(node)
                  ∨ ¬CheckPhase2a(node) ∧ CheckPhase2b(node)
                ∧ commonCoin' = [commonCoin EXCEPT ![round] = estimatesAtRound[round][node]]
          ∨ ∧ ∀ node ∈ NODES \ FAULTY_NODES :
                ∧ WaitForProposals(node)
                ∧ ¬CheckPhase2a(node)
                ∧ ¬CheckPhase2b(node)
            ∧ ∃ coinValue ∈ DecisionValues :   All nodes estimate both values based on the Common Coin.
                commonCoin' = [commonCoin EXCEPT ![round] = coinValue]
    ∧ UNCHANGED ⟨estimatesAtRound, proposalsAtRound, decisions, rounds⟩
```

Consensus property for *TLC* to check. Consensus is guaranteed only between non-faulty nodes. All decision values must be "0" or "1" and all non-faulty nodes must decide
on the same value.

```
ConsensusReachedByzantine ≜
    ∧ ∀ node ∈ NODES \ FAULTY_NODES : decisions[node] ∈ DecisionValues
    ∧ ∀ node1, node2 ∈ NODES \ FAULTY_NODES : decisions[node1] = decisions[node2]
```

Common Coin validity property for *TLC* to check. The *CC* is valid, if its actually set during every round.

```
CommonCoinValid ≜
    ∧ ∀ round ∈ NodeRounds : commonCoin[round] ∈ DecisionValues
```

The last state of the algorithm when consensus is reached. Used to stop the algorithm by omitting behaviors after consensus is reached.

```
LastState ≜
    ∧ ConsensusReachedByzantine

TypeOK ≜
    ∧   ∀ round ∈ NodeRounds : estimatesAtRound[round] ∈ AllNodeEstimateFunctions
    ∧   ∀ round ∈ NodeRounds : proposalsAtRound[round] ∈ NodeProposalFunctions
    ∧   ∀ node ∈ NODES : decisions[node] ∈ NodeDecisionValues
    ∧   ∀ node ∈ NODES : rounds[node] ∈ NodeRounds
    ∧   ∀ round ∈ NodeRounds : commonCoin[round] ∈ CommonCoinValues
```

If *CHECK_ALL_INITIAL_VALUES* is set to TRUE, *TLC* will check all possible estimate value combinations. Otherwise, initial estimates are defined by configuring what nodes estimate a "0" or a "1" value with the sets *ESTIMATE_ZERO* and *ESTIMATE_ONE*.
Estimates for all other rounds are set to "-1". Proposal message values for all rounds are set to "-1". Initial decisions are set to "-1" at the start to ensure *ConsensusReached* is not true
in the initial state.
Initially no Common Coins were tossed at any round yet.

```
Init ≜
    ∧ IF CHECK_ALL_INITIAL_VALUES
        THEN ∃ estimateFunction ∈ ValidNodeEstimateFunctions :
                estimatesAtRound = [round ∈ NodeRounds ↦ IF round = 1
                                                            THEN estimateFunction
                                                            ELSE [node ∈ NODES ↦ "-1"]]
        ELSE estimatesAtRound = [round ∈ NodeRounds ↦ IF round = 1
                                                            THEN [node ∈ NODES ↦
                                                                IF node ∈ ESTIMATE_ZERO
                                                                 THEN "0"
                                                                 ELSE "1"]
                                                            ELSE [node ∈ NODES ↦ "-1"]]
    ∧ proposalsAtRound = [round ∈ NodeRounds ↦ [node ∈ NODES ↦ "-1"]]
    ∧ decisions = [node ∈ NODES ↦ "-1"]
```

$\quad \land\ rounds = [node \in NODES \mapsto 1]$
$\quad \land\ commonCoin = [round \in NodeRounds \mapsto \text{"-1"}]$

$Next \triangleq$
$\quad \land\ \neg LastState$ Stops *TLC* when used with deadlock check enabled.
$\quad \land\ \lor\ \exists\, node \in NODES :$
$\qquad\qquad \lor\ Phase1(node)$
$\qquad\qquad \lor\ Phase2(node)$
$\qquad\qquad \lor\ SetNextRoundEstimate(node)$
$\qquad\ \ \lor\ TossCommonCoin$

$Spec \triangleq$
$\quad \land\ Init$
$\quad \land\ \Box[Next]_{vars}$
$\quad \land\ \forall\, node \in NODES :$
$\qquad \mathrm{WF}_{vars}(Phase1(node) \lor Phase2(node) \lor SetNextRoundEstimate(node))$
$\quad \land\ \mathrm{WF}_{vars}(TossCommonCoin)$

$ConsensusProperty \triangleq \Diamond\Box[ConsensusReachedByzantine]_{vars}$ Eventually, consensus will always be reached.
$CommonCoinProperty \triangleq \Diamond\Box[CommonCoinValid]_{vars}$ Eventually, the *CC* will be set for all rounds.

THEOREM $Spec \Rightarrow \Box\, TypeOK$ Checked by *TLC*.
PROOF OMITTED

THEOREM $Spec \Rightarrow ConsensusProperty$ Checked by *TLC*.
PROOF OMITTED

THEOREM $Spec \Rightarrow CommonCoinProperty$ Checked by *TLC*.
PROOF OMITTED

Figure 28. The BenOrAbstByzCC TLA^{+} specification

MODULE *BenOrAbstByzCCH*

This is a refinement of *BenOrMsgsByzCC*, *BenOrAbstByzCC*, and *BenOrMsgsByzCCH* with additional refinement history variables. Several modifications were required for *TLC* to successfully verify the refinement mappings *RefMsgs* :

– Added the *msgsHistory* and *estimateHistory* variables that correspond to the *msgs* and estimates variables from *BenOrMsgsByzCC*.

- Modified the specification so that proposal messages are added to the *msgsHistory* variable when they are sent. Also, faulty node proposals with "?" values are added to *msgsHistory* too. Likewise, report messages with estimate values are added in the action formula *SetNextRoundEstimate*.
- Made it so that whenever a node estimates a new value, then *estimateHistory* is also changed.
- Modified the *Init* formula to also set the *estimateHistory* values based on the initial estimates similarly to *BenOrMsgsByzCC*.
- Added actions for checking whether certain messages are present in the *msgsHistory* variable. Added corresponding checks for whether specific messages were sent by a node in phase 1, phase 2, and the *TossCommonCoin* action. This does not alter the state space in a way to make the mapping *RefAbstCC* incorrect, and is required for *TLC* to verify the mapping *RefMsgs*.
- Made it so that during the first phase of the first round, at least $N - F$ initial estimates are sent as report messages before continuing to the second phase.

The refinement mappings *RefMsgsH* and *RefAbstCC* are also provided and can be checked with *TLC* if put inside the *RefinementProperty* definition before '!*Spec*'.

EXTENDS TLC, $Integers$, $FiniteSets$

Use the *BenOrByzCC.cfg* file.

CONSTANTS $NODES$, $FAULTY\_NODES$, $ESTIMATE\_ZERO$, $ESTIMATE\_ONE$, $CHECK\_ALL\_INITIAL\_VALUES$

VARIABLE $estimatesAtRound$	Function that returns a function of all the node's estimates at the specified round.
VARIABLE $proposalsAtRound$	Function that returns a function of all the node's proposal values at the specified round.
VARIABLE $decisions$	Function that returns a given node's current decision value.
VARIABLE $rounds$	Function that returns a given node's current round.
VARIABLE $commonCoin$	Models the Common Coin.
VARIABLE $msgsHistory$	History variables for refinement.
VARIABLE $estimateHistory$	

$vars \triangleq \langle estimatesAtRound, proposalsAtRound, decisions, rounds, commonCoin, msgsHistory, estimateHistory \rangle$

For the *BenOr Byzantine* consensus protocol version, $N > 5 * F$ must be true. Also, F must be greater than 0, otherwise the $N - F$ check for proposal messages won't work.

ASSUME $\land (Cardinality(NODES) > 5 * Cardinality(FAULTY\_NODES))$
$\qquad\quad \land (Cardinality(FAULTY\_NODES) \geq 1)$

Nodes and their decisions must be specified.

ASSUME $\land NODES \neq \{\}$
$\qquad\quad \land ESTIMATE\_ONE \neq \{\}$
$\qquad\quad \land ESTIMATE\_ZERO \neq \{\}$

Decision values must be specified for all nodes.

ASSUME $Cardinality(ESTIMATE\_ONE) + Cardinality(ESTIMATE\_ZERO) = Cardinality(NODES)$

Numbers of nodes and faulty nodes.

$NumberOfNodes \triangleq Cardinality(NODES)$
$NumberOfFaultyNodes \triangleq Cardinality(FAULTY\_NODES)$

Only a limited number of rounds is modeled.

$NodeRounds \triangleq 1 \mathrel{..} 3$

Decision values of binary consensus.

$DecisionValues \triangleq \{\text{"0"}, \text{"1"}\}$

Set of all possible node decision values including the "-1" used for the initial state.

$NodeDecisionValues \triangleq DecisionValues \cup \{\text{"-1"}\}$

Set of possible proposal values.

$ProposalValues \triangleq \{\text{"?"}, \text{"0"}, \text{"1"}\}$

Set of all possible proposal message values including the "-1" used to indicate a proposal was not yet sent.

$ProposalMessageValues \triangleq ProposalValues \cup \{\text{"-1"}\}$

Set of all possible node estimate values. The "-1" is used to track whether a node has sent a report message or not.

$NodeEstimateValues \triangleq DecisionValues \cup \{\text{"-1"}\}$

Set of all functions that map nodes to their estimate values and "-1".

$AllNodeEstimateFunctions \triangleq [NODES \rightarrow NodeEstimateValues]$

Set of all functions that map nodes to their estimate values "0" or "1".

```
ValidNodeEstimateFunctions ≜ [NODES → DecisionValues]
```

Set of all functions that map nodes to their proposal message values.

```
NodeProposalFunctions ≜ [NODES → ProposalMessageValues]
```

Set of all possible Common Coin values. "-1" means that the common coin was not tossed yet for that round.

```
CommonCoinValues ≜ {"-1", "0", "1"}
```

Possible values and types of messages in the *BenOr* consensus algorithm.

```
MessageType ≜ {"report", "proposal"}
MessageValues ≜ {"0", "1", "?"}
```

Each message record also has a round number.

```
MessageRecordSet ≜ [Sender : NODES, Round : Nat, Type : MessageType, Value : MessageValues]
```

Used to print variable values. Must be used with ∧ and in states which are reached by *TLC*.

```
PrintVal(message, expression) ≜ Print(⟨message, expression⟩, TRUE)
```

Used to print the values of all variables.

```
PrintAllVariableValues ≜
    ∧ PrintVal("Estimates at round: ", estimatesAtRound)
    ∧ PrintVal("Proposals at round: ", proposalsAtRound)
    ∧ PrintVal("Decisions: ", decisions)
    ∧ PrintVal("Rounds: ", rounds)
    ∧ PrintVal("Common Coin:", commonCoin)
```

A set of all nodes other than the specified node.

```
AllOtherNodes(node) ≜
    NODES \ {node}
```

Returns TRUE if a node sent the specified message during the specified round.

```
DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue) ≜
    LET messageRecord ≜ [Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ messageValue]
    IN  messageRecord ∈ msgsHistory
```

Returns TRUE if a node sent a message of the specified type with any value ("0", "1", or "?") during the specified round.

```
DidNodeSendMessageOfType(sendingNode, nodeRound, messageType) ≜
    ∃ msgValue ∈ MessageValues :
      LET messageRecord ≜ [Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ msgValue]
      IN  messageRecord ∈ msgsHistory
```

Returns the number of messages sent by nodes other than *checkingNode* of the specified message type.

```
HowManyMessagesOfType(checkingNode, nodeRound, messageType) ≜
    Cardinality({node ∈ AllOtherNodes(checkingNode) :
        DidNodeSendMessageOfType(node, nodeRound, messageType)})
```

Adds a new message to the *msgsHistory* variable.

```
AddToMessageHistory(sendingNode, nodeRound, messageType, messageValue) ≜
    ∧ ¬DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue)
    ∧ msgsHistory′ =
          msgsHistory ∪ {[Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ messageValue]}
```

The number of node estimates of a given estimated value made by nodes other than the checking node in the same round.

```
HowManyEstimates(checkingNode, estimatedValue) ≜
    Cardinality({node ∈ NODES :
        ∧ node ≠ checkingNode
        ∧ estimatesAtRound[rounds[checkingNode]][node] = estimatedValue})
```

Sends a proposal message with a value that is in more than $(N + F)$ / 2 estimates.

```
SendProposalMessage(node) ≜
    ∃ estimateValue ∈ DecisionValues :
      ∧ HowManyEstimates(node, estimateValue) ∗ 2 > NumberOfNodes + NumberOfFaultyNodes
      ∧ proposalsAtRound′ = [proposalsAtRound EXCEPT ![rounds[node]][node] = estimateValue]
      ∧ AddToMessageHistory(node, rounds[node], "proposal", estimateValue)
```

Check if there are more than $(N + F)$ / 2 estimates of the same value.

```
$CheckPhase1a(node) \triangleq$
    $\exists estimateValue \in DecisionValues :$
        $HowManyEstimates(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes$
```

Wait for $N - F$ nodes with estimate values other than "-1" in the same round.

```
$WaitForReports(waitingNode) \triangleq$
    $Cardinality(\{node \in NODES :$
        $\land node \neq waitingNode$
        $\land estimatesAtRound[rounds[waitingNode]][node] \in DecisionValues\}) \geq NumberOfNodes - NumberOfFaultyNodes$
```

Wait for $N - F$ nodes to send report messages with their estimate values.

```
$WaitForReportMessages(node) \triangleq$
    $HowManyMessagesOfType(node, rounds[node], \text{"report"}) \geq NumberOfNodes - NumberOfFaultyNodes$
```

The first phase of the algorithm. All nodes check whether there are more than $(N + F)$ / 2 estimates with the same value v. If such a value exists, then v is sent to all other nodes in a proposal message. v is not necessarily the same as the node's current estimate value. A node can send
a value different from its own estimate value to other nodes in a proposal message.
If such a value doesn't exist, the node sends a proposal message with a "?" value. A faulty node always sends a proposal message with the "?" value.

```
$Phase1(node) \triangleq$
    $\land node \in NODES$
    $\land proposalsAtRound[rounds[node]][node] \notin ProposalValues$
    $\land WaitForReports(node)$
    $\land$ IF $rounds[node] = 1$   Send initial estimate values within report messages.
        THEN $WaitForReportMessages(node)$
        ELSE TRUE
    $\land DidNodeSendMessageOfType(node, rounds[node], \text{"report"})$
    $\land$ IF $\land CheckPhase1a(node)$
           $\land node \notin FAULTY\_NODES$
        THEN $SendProposalMessage(node)$
        ELSE $\land proposalsAtRound' = [proposalsAtRound$ EXCEPT $![rounds[node]][node] = \text{"?"}]$
             $\land AddToMessageHistory(node, rounds[node], \text{"proposal"}, \text{"?"})$
    $\land$ UNCHANGED $\langle estimatesAtRound, decisions, rounds, commonCoin, estimateHistory\rangle$
```

The number of node proposals of a given proposal value made by nodes other than the checking node in the same round.

```
$HowManyProposals(checkingNode, proposalValue) \triangleq$
    $Cardinality(\{node \in NODES :$
        $\land node \neq checkingNode$
        $\land proposalsAtRound[rounds[checkingNode]][node] = proposalValue\})$
```

Returns a set of proposal values "0" or "1" that are proposed by more than $(N + F)$ / 2 nodes.

```
$AtLeastMajorityProposals(node) \triangleq$
    $\{estimateValue \in DecisionValues :$
        $HowManyProposals(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes\}$
```

Returns a set of proposal values "0" or "1" that are proposed by at least $F + 1$ nodes.

```
$AtLeastNF1Proposals(node) \triangleq$
    $\{estimateValue \in DecisionValues :$
        $HowManyProposals(node, estimateValue) \geq NumberOfFaultyNodes + 1\}$
```

Check for $F + 1$ or more proposal messages with the same value.

```
$CheckPhase2a(node) \triangleq$
    $AtLeastMajorityProposals(node) \neq \{\}$
```

Check for $F + 1$ or more proposal messages with the same value.

```
$CheckPhase2b(node) \triangleq$
    $AtLeastNF1Proposals(node) \neq \{\}$
```

Decide an estimate value that has been proposed by more than $(N + F)$ / 2 nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$DecidePhase2a(node) \triangleq$
    $\exists estimateValue \in AtLeastMajorityProposals(node) :$
        $decisions' = [decisions$ EXCEPT $![node] = estimateValue]$
```

Estimate a proposal value that has been proposed by at least $F + 1$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$EstimatePhase2b(node) \triangleq$
    $\exists\, estimateValue \in AtLeastNF1Proposals(node):$
        $\land\ estimatesAtRound' = [estimatesAtRound\ \text{EXCEPT}\ ![rounds[node]][node] = estimateValue]$
        $\land\ estimateHistory' = [estimateHistory\ \text{EXCEPT}\ ![node] = estimateValue]$
```

Pick am estimate value from "0" or "1" for a node based on the common coin of the current round.

```
$EstimatePhase2c(node) \triangleq$
    $\land\ commonCoin[rounds[node]] \neq$ "-1"
    $\land\ estimatesAtRound' = [estimatesAtRound\ \text{EXCEPT}\ ![rounds[node]][node] = commonCoin[rounds[node]]]$
    $\land\ estimateHistory' = [estimateHistory\ \text{EXCEPT}\ ![node] = commonCoin[rounds[node]]]$
```

Wait for $N - F$ proposal messages with values other than "-1" in the same round.

```
$WaitForProposals(waitingNode) \triangleq$
    $Cardinality(\{node \in NODES :$
        $\land\ node \neq waitingNode$
        $\land\ proposalsAtRound[rounds[waitingNode]][node] \in ProposalValues\}) \geq NumberOfNodes - NumberOfFaultyNodes$
```

Second phase of the *BenOr* consensus algorithm. All nodes check whether there have been more than $(N + F)\ /\ 2$ proposal messages with the same value "0" or "1". If there were that many messages, each node updates their decision and estimate values to match the value in the proposal messages.
Otherwise, if there are at least $F + 1$ received proposal messages with the same value value "0" or "1", the value is set as the new estimate for the receiving node.
In all other cases, nodes estimate a random value of "0" or "1" for the next round.

```
$Phase2(node) \triangleq$
    $\land\ node \in NODES$
    $\land\ WaitForProposals(node)$
    $\land\ DidNodeSendMessageOfType(node, rounds[node],$ "proposal"$)$
    $\land\ \text{IF}\ CheckPhase2a(node) \land CheckPhase2b(node)$
        $\text{THEN}\ DecidePhase2a(node) \land EstimatePhase2b(node)$   If there are more than $(N + F)\ /\ 2$ valid proposals . .
        $\text{ELSE}\ \text{IF}\ \neg CheckPhase2a(node) \land CheckPhase2b(node)$
                $\text{THEN}\ EstimatePhase2b(node) \land \text{UNCHANGED}\ \langle decisions \rangle$   If there are at least $F + 1$ valid proposals.
                $\text{ELSE}\ \text{IF}\ \neg CheckPhase2a(node) \land \neg CheckPhase2b(node)$
                        $\text{THEN}\ EstimatePhase2c(node) \land \text{UNCHANGED}\ \langle decisions \rangle$   Estimates are randomized.
                        $\text{ELSE}\ \text{FALSE}$   No other cases possible.
    $\land\ rounds' = [rounds\ \text{EXCEPT}\ ![node] = @ + 1]$
    $\land\ \text{UNCHANGED}\ \langle proposalsAtRound, commonCoin, msgsHistory \rangle$
```

Set the node's next round's estimate to the previous round's estimate value so that other nodes are aware that the node has entered the next round. In the previous round the estimate will be some valid value other than "-1".

```
$SetNextRoundEstimate(node) \triangleq$
    $\land\ node \in NODES$
    $\land\ \text{IF}\ rounds[node] = 1 \land estimatesAtRound[rounds[node]][node] \neq$ "-1"
        $\text{THEN}\ \land\ AddToMessageHistory(node, rounds[node],$ "report"$, estimatesAtRound[rounds[node]][node])$
              $\land\ \text{UNCHANGED}\ \langle estimatesAtRound \rangle$
        $\text{ELSE}\ \land\ estimatesAtRound[rounds[node]][node] =$ "-1"
              $\land\ estimatesAtRound' =$
                    $[estimatesAtRound\ \text{EXCEPT}\ ![rounds[node]][node] = estimatesAtRound[rounds[node] - 1][node]]$
              $\land\ AddToMessageHistory(node, rounds[node],$ "report"$, estimatesAtRound[rounds[node] - 1][node])$
    $\land\ \text{UNCHANGED}\ \langle proposalsAtRound, decisions, rounds, commonCoin, estimateHistory \rangle$
```

Set the Common Coin value for the current round. If in the round at least 1 node has already estimated some value non-randomly, then that round's common coin
value will be set to the node's estimate value (the Common Coin was lucky).
Otherwise, the Common Coin value is set to "0" or "1" and *TLC* will check behaviors when all nodes randomly estimate "0" or "1" together.

```
$TossCommonCoin \triangleq$
    $\land\ \exists\, round \in NodeRounds :$
        $\land\ commonCoin[round] =$ "-1"
        $\land\ \lor\ \exists\, node \in NODES \setminus FAULTY\_NODES :$   If at least 1 node already estimated some value in phase $2b$.
                $\land\ estimatesAtRound[round][node] \neq$ "-1"
                $\land\ WaitForProposals(node)$
                $\land\ DidNodeSendMessageOfType(node, rounds[node],$ "proposal"$)$
                $\land\ \lor\ CheckPhase2a(node) \land CheckPhase2b(node)$
                ∨ ¬CheckPhase2a(node) ∧ CheckPhase2b(node)
              ∧ commonCoin′ = [commonCoin EXCEPT ![round] = estimatesAtRound[round][node]]
          ∨ ∧ ∀ node ∈ NODES \ FAULTY_NODES :
                ∧ WaitForProposals(node)
                ∧ DidNodeSendMessageOfType(node, rounds[node], “proposal”)
                ∧ ¬CheckPhase2a(node)
                ∧ ¬CheckPhase2b(node)
            ∧ ∃ coinValue ∈ DecisionValues :   All nodes estimate both values based on the Common Coin.
              commonCoin′ = [commonCoin EXCEPT ![round] = coinValue]
    ∧ UNCHANGED ⟨estimatesAtRound, proposalsAtRound, decisions, rounds, msgsHistory, estimateHistory⟩
```

Consensus property for *TLC* to check. Consensus is guaranteed only between non-faulty nodes. All decision values must be “0” or “1” and all non-faulty nodes must decide
on the same value.

```
ConsensusReachedByzantine ≜
    ∧ ∀ node ∈ NODES \ FAULTY_NODES : decisions[node] ∈ DecisionValues
    ∧ ∀ node1, node2 ∈ NODES \ FAULTY_NODES : decisions[node1] = decisions[node2]
```

Common Coin validity property for *TLC* to check. The *CC* is valid, if its actually set during every round.

```
CommonCoinValid ≜
    ∧ ∀ round ∈ NodeRounds : commonCoin[round] ∈ DecisionValues
```

The last state of the algorithm when consensus is reached. Used to stop the algorithm by omitting behaviors after consensus is reached.

```
LastState ≜
    ∧ ConsensusReachedByzantine

TypeOK ≜
    ∧   ∀ round ∈ NodeRounds : estimatesAtRound[round] ∈ AllNodeEstimateFunctions
    ∧   ∀ round ∈ NodeRounds : proposalsAtRound[round] ∈ NodeProposalFunctions
    ∧   ∀ node ∈ NODES : decisions[node] ∈ NodeDecisionValues
    ∧   ∀ node ∈ NODES : rounds[node] ∈ NodeRounds
    ∧   ∀ round ∈ NodeRounds : commonCoin[round] ∈ CommonCoinValues
    ∧   msgsHistory ⊆ MessageRecordSet
```

If *CHECK_ALL_INITIAL_VALUES* is set to TRUE, *TLC* will check all possible estimate value combinations. Otherwise, initial estimates
are defined by configuring what nodes estimate a “0” or a “1” value with the sets *ESTIMATE_ZERO* and *ESTIMATE_ONE*.
Estimates for all other rounds are set to “-1”. Proposal message values for all rounds are set to “-1”. Initial decisions are set to “-1” at the start to ensure *ConsensusReached* is not true
in the initial state.
Initially no Common Coins were tossed at any round yet.

```
Init ≜
    ∧ IF CHECK_ALL_INITIAL_VALUES
       THEN ∃ estimateFunction   ∈ ValidNodeEstimateFunctions :
              ∧ estimatesAtRound = [round ∈ NodeRounds ↦ IF round = 1
                                                          THEN estimateFunction
                                                          ELSE [node ∈ NODES ↦ “-1”]]
              ∧ estimateHistory = estimateFunction
       ELSE   ∧ estimatesAtRound = [round ∈ NodeRounds ↦ IF round = 1
                                                          THEN [node ∈ NODES ↦
                                                              IF node ∈ ESTIMATE_ZERO
                                                               THEN “0”
                                                               ELSE “1”]
                                                          ELSE [node ∈ NODES ↦ “-1”]]
              ∧ estimateHistory = [node ∈ NODES ↦ IF node ∈ ESTIMATE_ZERO THEN “0” ELSE “1”]
    ∧ proposalsAtRound = [round ∈ NodeRounds ↦ [node ∈ NODES ↦ “-1”]]
    ∧ decisions = [node ∈ NODES ↦ “-1”]
    ∧ rounds = [node ∈ NODES ↦ 1]
    ∧ commonCoin = [round ∈ NodeRounds ↦ “-1”]
    ∧ msgsHistory = {}

Next ≜
    ∧ ¬LastState  Stops TLC when used with deadlock check enabled.
    ∧ ∨ ∃ node ∈ NODES :
        ∨ $Phase1(node)$
        ∨ $Phase2(node)$
        ∨ $SetNextRoundEstimate(node)$
    ∨ $TossCommonCoin$

$Spec$ $\triangleq$
    ∧ $Init$
    ∧ $\Box[Next]_{vars}$
    ∧ $\forall\, node \in NODES$ :
        $\mathrm{WF}_{vars}(Phase1(node) \lor Phase2(node) \lor SetNextRoundEstimate(node))$
    ∧ $\mathrm{WF}_{vars}(TossCommonCoin)$
```

VERY IMPORTANT: when checking the *RefinementProperty*, all INSTANCE statements in the other refined history variable specification *.tla* file must be commented out to avoid circular dependencies.
Also, the *RefinementProperty* definition must be commented out as well since the INSTANCE statements would not be defined in such a case.

```
$RefMsgsH$ $\triangleq$
    INSTANCE $BenOrMsgsByzCCH$
     WITH $msgs \leftarrow msgsHistory$,
          $estimates \leftarrow estimateHistory$,
          $estimateHistory \leftarrow estimatesAtRound$,
          $proposalHistory \leftarrow proposalsAtRound$

$RefMsgs$ $\triangleq$
    INSTANCE $BenOrMsgsByzCC$
     WITH $msgs \leftarrow msgsHistory$,
          $estimates \leftarrow estimateHistory$

$RefAbstCC$ $\triangleq$
    INSTANCE $BenOrAbstByzCC$

$RefinementProperty$ $\triangleq$      Can be uncommented here and in the BenOrByzCC.cfg file to check refinement.
    ∧ $RefMsgs!Spec$
    ∧ $RefMsgsH!Spec$
    ∧ $RefAbstCC!Spec$
$ConsensusProperty$ $\triangleq$ $\Diamond\Box[ConsensusReachedByzantine]_{vars}$   Eventually, consensus will always be reached.
$CommonCoinProperty$ $\triangleq$ $\Diamond\Box[CommonCoinValid]_{vars}$   Eventually, the CC will be set for all rounds.
```

```
THEOREM $Spec \Rightarrow \Box TypeOK$   Checked by TLC.
PROOF OMITTED

THEOREM $Spec \Rightarrow ConsensusProperty$   Checked by TLC.
PROOF OMITTED

THEOREM $Spec \Rightarrow CommonCoinProperty$   Checked by TLC.
PROOF OMITTED
```

Figure 29. The BenOrAbstByzCCH TLA^{+} specification

MODULE *BenOrAbstByzCCL*

This specification is a refinement of *BenOrAbstByzCC* that contains a node action for tossing the common coin. Now instead of the coin being tossed by the system, some node tosses the coin.
The refinement mapping *RefAbstCC* is also included in the specification and can be checked with *TLC*. The specification contains fewer distinct states compared to *BenOrAbstByzCC* but as the number
of nodes increase the amount of total states becomes greater than *BenOrAbstByzCC*.

EXTENDS *TLC*, *Integers*, *FiniteSets*

Use the *BenOrByzCC.cfg* file.
CONSTANTS *NODES*, *FAULTY_NODES*, *ESTIMATE_ZERO*, *ESTIMATE_ONE*, *CHECK_ALL_INITIAL_VALUES*

VARIABLE *estimatesAtRound* Function that returns a function of all the node's estimates at the specified round.
VARIABLE *proposalsAtRound* Function that returns a function of all the node's proposal values at the specified round.
VARIABLE *decisions* Function that returns a given node's current decision value.
VARIABLE *rounds* Function that returns a given node's current round.
VARIABLE *commonCoin* Models the Common Coin.

$vars \triangleq \langle estimatesAtRound, proposalsAtRound, decisions, rounds, commonCoin \rangle$

For the *BenOr Byzantine* consensus protocol version, $N > 5 * F$ must be true. Also, F must be greater than 0, otherwise the $N - F$ check for proposal messages won't work.
ASSUME $\land\ (Cardinality(NODES) > 5 * Cardinality(FAULTY\_NODES))$
$\qquad\quad \land\ (Cardinality(FAULTY\_NODES) \geq 1)$

Nodes and their decisions must be specified.
ASSUME $\land\ NODES \neq \{\}$
$\qquad\quad \land\ ESTIMATE\_ONE \neq \{\}$
$\qquad\quad \land\ ESTIMATE\_ZERO \neq \{\}$

Decision values must be specified for all nodes.
ASSUME $Cardinality(ESTIMATE\_ONE) + Cardinality(ESTIMATE\_ZERO) = Cardinality(NODES)$

Numbers of nodes and faulty nodes.
$NumberOfNodes \triangleq Cardinality(NODES)$
$NumberOfFaultyNodes \triangleq Cardinality(FAULTY\_NODES)$

Only a limited number of rounds is modeled.
$NodeRounds \triangleq 1 \mathrel{..} 3$

Decision values of binary consensus.
$DecisionValues \triangleq \{\text{"0"}, \text{"1"}\}$

Set of all possible node decision values including the "-1" used for the initial state.
$NodeDecisionValues \triangleq DecisionValues \cup \{\text{"-1"}\}$

Set of possible proposal values.
$ProposalValues \triangleq \{\text{"?"}, \text{"0"}, \text{"1"}\}$

Set of all possible proposal message values including the "-1" used to indicate a proposal was not yet sent.
$ProposalMessageValues \triangleq ProposalValues \cup \{\text{"-1"}\}$

Set of all possible node estimate values. The "-1" is used to track whether a node has sent a report message or not.
$NodeEstimateValues \triangleq DecisionValues \cup \{\text{"-1"}\}$

Set of all functions that map nodes to their estimate values and "-1".
$AllNodeEstimateFunctions \triangleq [NODES \rightarrow NodeEstimateValues]$

Set of all functions that map nodes to their estimate values "0" or "1".
$ValidNodeEstimateFunctions \triangleq [NODES \rightarrow DecisionValues]$

Set of all functions that map nodes to their proposal message values.
$NodeProposalFunctions \triangleq [NODES \rightarrow ProposalMessageValues]$

Set of all possible Common Coin values. "-1" means that the common coin was not tossed yet for that round.
$CommonCoinValues \triangleq \{\text{"-1"}, \text{"0"}, \text{"1"}\}$

Used to print variable values. Must be used with $\land$ and in states which are reached by *TLC*.

```
$PrintVal(message, expression) \triangleq Print(\langle message, expression \rangle, \text{TRUE})$
```

Used to print the values of all variables.

```
$PrintAllVariableValues \triangleq$
    $\land PrintVal($“Estimates at round: ”$, estimatesAtRound)$
    $\land PrintVal($“Proposals at round: ”$, proposalsAtRound)$
    $\land PrintVal($“Decisions: ”$, decisions)$
    $\land PrintVal($“Rounds: ”$, rounds)$
    $\land PrintVal($“Common Coin:”$, commonCoin)$
```

A set of all nodes other than the specified node.

```
$AllOtherNodes(node) \triangleq$
    $NODES \setminus \{node\}$
```

The number of node estimates of a given estimated value made by nodes other than the checking node in the same round.

```
$HowManyEstimates(checkingNode, estimatedValue) \triangleq$
    $Cardinality(\{node \in NODES :$
        $\land node \neq checkingNode$
        $\land estimatesAtRound[rounds[checkingNode]][node] = estimatedValue\})$
```

Sends a proposal message with a value that is in more than $(N + F) \ / \ 2$ estimates.

```
$SendProposalMessage(node) \triangleq$
   $\exists estimateValue \in DecisionValues :$
      $\land HowManyEstimates(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes$
      $\land proposalsAtRound' = [proposalsAtRound \text{ EXCEPT } ![rounds[node]][node] = estimateValue]$
```

Check if there are more than $(N + F) \ / \ 2$ estimates of the same value.

```
$CheckPhase1a(node) \triangleq$
   $\exists estimateValue \in DecisionValues :$
      $HowManyEstimates(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes$
```

Wait for $N - F$ nodes with estimate values other than “-1” in the same round.

```
$WaitForReports(waitingNode) \triangleq$
    $Cardinality(\{node \in NODES :$
        $\land node \neq waitingNode$
        $\land estimatesAtRound[rounds[waitingNode]][node] \in DecisionValues\}) \geq NumberOfNodes - NumberOfFaultyNodes$
```

The first phase of the algorithm. All nodes check whether there are more than $(N + F) \ / \ 2$ estimates with the same value v. If such a value exists, then v is sent to all other nodes in a proposal message. v is not necessarily the same as the node’s current estimate value. A node can send
a value different from its own estimate value to other nodes in a proposal message.
If such a value doesn’t exist, the node sends a proposal message with a “?” value. A faulty node always sends a proposal message with the “?” value.

```
$Phase1(node) \triangleq$
    $\land node \in NODES$
    $\land proposalsAtRound[rounds[node]][node] \notin ProposalValues$
    $\land WaitForReports(node)$
    $\land \text{IF } \land CheckPhase1a(node)$
             $\land node \notin FAULTY\_NODES$
       $\text{THEN } SendProposalMessage(node)$
       $\text{ELSE } proposalsAtRound' = [proposalsAtRound \text{ EXCEPT } ![rounds[node]][node] = \text{“?”}]$
    $\land \text{UNCHANGED } \langle estimatesAtRound, decisions, rounds, commonCoin \rangle$
```

The number of node proposals of a given proposal value made by nodes other than the checking node in the same round.

```
$HowManyProposals(checkingNode, proposalValue) \triangleq$
    $Cardinality(\{node \in NODES :$
        $\land node \neq checkingNode$
        $\land proposalsAtRound[rounds[checkingNode]][node] = proposalValue\})$
```

Returns a set of proposal values “0” or “1” that are proposed by more than $(N + F) \ / \ 2$ nodes.

```
$AtLeastMajorityProposals(node) \triangleq$
    $\{estimateValue \in DecisionValues :$
        $HowManyProposals(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes\}$
```

Returns a set of proposal values “0” or “1” that are proposed by at least $F + 1$ nodes.

```
AtLeastNF1Proposals(node) ≜
    {estimateValue ∈ DecisionValues :
        HowManyProposals(node, estimateValue) ≥ NumberOfFaultyNodes + 1}
```

Check for $F + 1$ or more proposal messages with the same value.

```
CheckPhase2a(node) ≜
    AtLeastMajorityProposals(node) ≠ {}
```

Check for $F + 1$ or more proposal messages with the same value.

```
CheckPhase2b(node) ≜
    AtLeastNF1Proposals(node) ≠ {}
```

Decide an estimate value that has been proposed by more than $(N + F)$ / 2 nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
DecidePhase2a(node) ≜
    ∃ estimateValue ∈ AtLeastMajorityProposals(node) :
        decisions' = [decisions EXCEPT ![node] = estimateValue]
```

Estimate a proposal value that has been proposed by at least $F + 1$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
EstimatePhase2b(node) ≜
    ∃ estimateValue ∈ AtLeastNF1Proposals(node) :
        estimatesAtRound' = [estimatesAtRound EXCEPT ![rounds[node]][node] = estimateValue]
```

Pick an estimate value from "0" or "1" for a node based on the common coin of the current round.

```
EstimatePhase2c(node) ≜
    ∧ commonCoin[rounds[node]] ≠ "-1"
    ∧ estimatesAtRound' = [estimatesAtRound EXCEPT ![rounds[node]][node] = commonCoin[rounds[node]]]
```

Wait for $N - F$ proposal messages with values other than "-1" in the same round.

```
WaitForProposals(waitingNode) ≜
    Cardinality({node ∈ NODES :
        ∧ node ≠ waitingNode
        ∧ proposalsAtRound[rounds[waitingNode]][node] ∈ ProposalValues}) ≥ NumberOfNodes − NumberOfFaultyNodes
```

Second phase of the *BenOr* consensus algorithm. All nodes check whether there have been more than $(N + F)$ / 2 proposal messages with the same value "0" or "1". If there were that many messages, each node updates their decision and estimate values to match the value in the proposal messages.
Otherwise, if there are at least $F + 1$ received proposal messages with the same value value "0" or "1", the value is set as the new estimate for the receiving node.
In all other cases, nodes estimate a random value of "0" or "1" for the next round.

```
Phase2(node) ≜
    ∧ node ∈ NODES
    ∧ WaitForProposals(node)
    ∧ IF CheckPhase2a(node) ∧ CheckPhase2b(node)
        THEN DecidePhase2a(node) ∧ EstimatePhase2b(node)  If there are more than (N + F) / 2 valid proposals.
        ELSE IF ¬CheckPhase2a(node) ∧ CheckPhase2b(node)
                THEN EstimatePhase2b(node) ∧ UNCHANGED ⟨decisions⟩  If there are at least F + 1 valid proposals.
                ELSE IF ¬CheckPhase2a(node) ∧ ¬CheckPhase2b(node)
                        THEN EstimatePhase2c(node) ∧ UNCHANGED ⟨decisions⟩  Estimates are randomized.
                        ELSE FALSE  No other cases possible.
    ∧ rounds' = [rounds EXCEPT ![node] = @ + 1]
    ∧ UNCHANGED ⟨proposalsAtRound, commonCoin⟩
```

Set the node's next round's estimate to the previous round's estimate value so that other nodes are aware that the node has entered the next round. In the previous round the estimate will be some valid value other than "-1".

```
SetNextRoundEstimate(node) ≜
    ∧ node ∈ NODES
    ∧ estimatesAtRound[rounds[node]][node] = "-1"
    ∧ estimatesAtRound' = [estimatesAtRound EXCEPT ![rounds[node]][node] = estimatesAtRound[rounds[node] − 1][node]]
    ∧ UNCHANGED ⟨proposalsAtRound, decisions, rounds, commonCoin⟩
```

A node sets the Common Coin value for the current round the node is in. If in the round at least 1 node has already estimated some value non-randomly, then that round's common coin
value will be set to the node's estimate value (the Common Coin was lucky).

Otherwise, the Common Coin value is set to "0" or "1" and *TLC* will check behaviors when all nodes randomly estimate "0" or "1" together.

```
$TossCommonCoin(settingNode) \triangleq$
    $\land\ commonCoin[rounds[settingNode]] = \text{"-1"}$
    $\land\ \lor\ \exists\, node \in NODES \setminus FAULTY\_NODES :$   If at least 1 node already estimated some value in phase 2b.
              $\land\ estimatesAtRound[rounds[settingNode]][node] \neq \text{"-1"}$
              $\land\ WaitForProposals(node)$
              $\land\ \lor\ CheckPhase2a(node) \land CheckPhase2b(node)$
                 $\lor\ \neg CheckPhase2a(node) \land CheckPhase2b(node)$
              $\land\ commonCoin' =$
                 $[commonCoin\ \text{EXCEPT}\ ![rounds[settingNode]] = estimatesAtRound[rounds[settingNode]][node]]$
       $\lor\ \land\ \forall\, node \in NODES \setminus FAULTY\_NODES :$
              $\land\ WaitForProposals(node)$
              $\land\ \neg CheckPhase2a(node)$
              $\land\ \neg CheckPhase2b(node)$
         $\land\ \exists\, coinValue \in DecisionValues :$   All nodes estimate both values based on the Common Coin.
              $commonCoin' = [commonCoin\ \text{EXCEPT}\ ![rounds[settingNode]] = coinValue]$
    $\land\ \text{UNCHANGED}\ \langle estimatesAtRound, proposalsAtRound, decisions, rounds\rangle$
```

Consensus property for *TLC* to check. Consensus is guaranteed only between non-faulty nodes. All decision values must be "0" or "1" and all non-faulty nodes must decide
on the same value.

```
$ConsensusReachedByzantine \triangleq$
    $\land\ \forall\, node \in NODES \setminus FAULTY\_NODES : decisions[node] \in DecisionValues$
    $\land\ \forall\, node1, node2 \in NODES \setminus FAULTY\_NODES : decisions[node1] = decisions[node2]$
```

Common Coin validity property for *TLC* to check. The *CC* is valid, if its actually set during every round.

```
$CommonCoinValid \triangleq$
    $\land\ \forall\, round \in NodeRounds : commonCoin[round] \in DecisionValues$
```

The last state of the algorithm when consensus is reached. Used to stop the algorithm by omitting behaviors after consensus is reached.

```
$LastState \triangleq$
    $\land\ ConsensusReachedByzantine$

$TypeOK \triangleq$
    $\land\ \forall\, round \in NodeRounds : estimatesAtRound[round] \in AllNodeEstimateFunctions$
    $\land\ \forall\, round \in NodeRounds : proposalsAtRound[round] \in NodeProposalFunctions$
    $\land\ \forall\, node \in NODES : decisions[node] \in NodeDecisionValues$
    $\land\ \forall\, node \in NODES : rounds[node] \in NodeRounds$
    $\land\ \forall\, round \in NodeRounds : commonCoin[round] \in CommonCoinValues$
```

If *CHECK_ALL_INITIAL_VALUES* is set to TRUE, *TLC* will check all possible estimate value combinations. Otherwise, initial estimates are defined by configuring what nodes estimate a "0" or a "1" value with the sets *ESTIMATE_ZERO* and *ESTIMATE_ONE*.
Estimates for all other rounds are set to "-1". Proposal message values for all rounds are set to "-1". Initial decisions are set to "-1" at the start to ensure *ConsensusReached* is not true
in the initial state.
Initially no Common Coins were tossed at any round yet.

```
$Init \triangleq$
    $\land\ \text{IF}\ CHECK\_ALL\_INITIAL\_VALUES$
        $\text{THEN}\ \exists\, estimateFunction \in ValidNodeEstimateFunctions :$
              $estimatesAtRound = [round \in NodeRounds \mapsto \text{IF}\ round = 1$
                                                        $\text{THEN}\ estimateFunction$
                                                        $\text{ELSE}\ [node \in NODES \mapsto \text{"-1"}]]$
        $\text{ELSE}\ estimatesAtRound = [round \in NodeRounds \mapsto \text{IF}\ round = 1$
                                                        $\text{THEN}\ [node \in NODES \mapsto$
                                                           $\text{IF}\ node \in ESTIMATE\_ZERO$
                                                            $\text{THEN}\ \text{"0"}$
                                                            $\text{ELSE}\ \text{"1"}]$
                                                        $\text{ELSE}\ [node \in NODES \mapsto \text{"-1"}]]$
    $\land\ proposalsAtRound = [round \in NodeRounds \mapsto [node \in NODES \mapsto \text{"-1"}]]$
    $\land\ decisions = [node \in NODES \mapsto \text{"-1"}]$
    $\land\ rounds = [node \in NODES \mapsto 1]$
    $\land\ commonCoin = [round \in NodeRounds \mapsto \text{"-1"}]$
```

```
Next ≜
    ∃ node ∈ NODES :
        ∧ ¬LastState   Stops TLC when used with deadlock check enabled.
        ∧ ∨ Phase1(node)
          ∨ Phase2(node)
          ∨ TossCommonCoin(node)
          ∨ SetNextRoundEstimate(node)

Spec ≜
    ∧ Init
    ∧ □[Next]_vars
    ∧ ∀ node ∈ NODES :
        WF_vars(Phase1(node) ∨ Phase2(node) ∨ TossCommonCoin(node) ∨ SetNextRoundEstimate(node))

RefAbstCC ≜
    INSTANCE BenOrAbstByzCC

 RefinementProperty ≜ RefAbstCC!Spec \ *  Can be uncommented here and in the BenOrByzCC.cfg file to check refinement.
ConsensusProperty ≜ ◇□[ConsensusReachedByzantine]_vars   Eventually, consensus will always be reached.
CommonCoinProperty ≜ ◇□[CommonCoinValid]_vars   Eventually, the CC will be set for all rounds.
├──────────────────────────────────────────────────────────────────────────────┤
THEOREM Spec ⇒ □TypeOK   Checked by TLC.
PROOF OMITTED
├──────────────────────────────────────────────────────────────────────────────┤
THEOREM Spec ⇒ ConsensusProperty   Checked by TLC.
PROOF OMITTED
├──────────────────────────────────────────────────────────────────────────────┤
THEOREM Spec ⇒ CommonCoinProperty   Checked by TLC.
PROOF OMITTED
└──────────────────────────────────────────────────────────────────────────────┘
```

Figure 30. The BenOrAbstByzCCL TLA$^+$ specification

MODULE *BenOrAbstByzCCLH*

This is a refinement of *BenOrMsgsByzCCL*, *BenOrAbstByzCCL*, and *BenOrMsgsByzCCLH* with additional refinement history variables. Several modifications were required for *TLC* to successfully verify the refinement mappings *RefMsgs* :

– Added the *msgsHistory* and *estimateHistory* variables that correspond to the *msgs* and estimates variables from *BenOrMsgsByzCCL*.

- Modified the specification so that proposal messages are added to the *msgsHistory* variable when they are sent. Also, faulty node proposals with "?" values are added to *msgsHistory* too. Likewise, report messages with estimate values are added in the action formula *SetNextRoundEstimate*.
- Made it so that whenever a node estimates a new value, then *estimateHistory* is also changed.
- Modified the *Init* formula to also set the *estimateHistory* values based on the initial estimates similarly to *BenOrMsgsByzCCL*.
- Added actions for checking whether certain messages are present in the *msgsHistory* variable. Added corresponding checks for whether specific messages were sent by a node in phase 1, phase 2, and the *TossCommonCoin* action. This does not alter the state space in a way to make the mapping *RefAbstCCL* incorrect, and is required for *TLC* to verify the mapping *RefMsgs*.
- Made it so that during the first phase of the first round, at least $N - F$ initial estimates are sent as report messages before continuing to the second phase.

The refinement mappings *RefMsgsH* and *RefAbstCCL* are also provided and can be checked with *TLC* if put inside the *RefinementProperty* definition before '!*Spec*'.

EXTENDS *TLC*, *Integers*, *FiniteSets*

Use the *BenOrByzCC.cfg* file.

CONSTANTS *NODES*, *FAULTY_NODES*, *ESTIMATE_ZERO*, *ESTIMATE_ONE*, *CHECK_ALL_INITIAL_VALUES*

VARIABLE *estimatesAtRound*	Function that returns a function of all the node's estimates at the specified round.
VARIABLE *proposalsAtRound*	Function that returns a function of all the node's proposal values at the specified round.
VARIABLE *decisions*	Function that returns a given node's current decision value.
VARIABLE *rounds*	Function that returns a given node's current round.
VARIABLE *commonCoin*	Models the Common Coin.
VARIABLE *msgsHistory*	History variables for refinement.
VARIABLE *estimateHistory*	

$vars \triangleq \langle estimatesAtRound, proposalsAtRound, decisions, rounds, commonCoin, msgsHistory, estimateHistory \rangle$

For the *BenOr Byzantine* consensus protocol version, $N > 5 * F$ must be true. Also, F must be greater than 0, otherwise the $N - F$ check for proposal messages won't work.

ASSUME $\land (Cardinality(NODES) > 5 * Cardinality(FAULTY\_NODES))$
$\quad\quad\land (Cardinality(FAULTY\_NODES) \geq 1)$

Nodes and their decisions must be specified.

ASSUME $\land NODES \neq \{\}$
$\quad\quad\land ESTIMATE\_ONE \neq \{\}$
$\quad\quad\land ESTIMATE\_ZERO \neq \{\}$

Decision values must be specified for all nodes.

ASSUME $Cardinality(ESTIMATE\_ONE) + Cardinality(ESTIMATE\_ZERO) = Cardinality(NODES)$

Numbers of nodes and faulty nodes.

$NumberOfNodes \triangleq Cardinality(NODES)$
$NumberOfFaultyNodes \triangleq Cardinality(FAULTY\_NODES)$

Only a limited number of rounds is modeled.

$NodeRounds \triangleq 1 \,..\, 3$

Decision values of binary consensus.

$DecisionValues \triangleq \{\text{"0"}, \text{"1"}\}$

Set of all possible node decision values including the "-1" used for the initial state.

$NodeDecisionValues \triangleq DecisionValues \cup \{\text{"-1"}\}$

Set of possible proposal values.

$ProposalValues \triangleq \{\text{"?"}, \text{"0"}, \text{"1"}\}$

Set of all possible proposal message values including the "-1" used to indicate a proposal was not yet sent.

$ProposalMessageValues \triangleq ProposalValues \cup \{\text{"-1"}\}$

Set of all possible node estimate values. The "-1" is used to track whether a node has sent a report message or not.

$NodeEstimateValues \triangleq DecisionValues \cup \{\text{"-1"}\}$

Set of all functions that map nodes to their estimate values and "-1".

$AllNodeEstimateFunctions \triangleq [NODES \rightarrow NodeEstimateValues]$

Set of all functions that map nodes to their estimate values "0" or "1".

```
ValidNodeEstimateFunctions ≜ [NODES → DecisionValues]
```

Set of all functions that map nodes to their proposal message values.

```
NodeProposalFunctions ≜ [NODES → ProposalMessageValues]
```

Set of all possible Common Coin values. "-1" means that the common coin was not tossed yet for that round.

```
CommonCoinValues ≜ {"-1", "0", "1"}
```

Possible values and types of messages in the *BenOr* consensus algorithm.

```
MessageType ≜ {"report", "proposal"}
MessageValues ≜ {"0", "1", "?"}
```

Each message record also has a round number.

```
MessageRecordSet ≜ [Sender : NODES, Round : Nat, Type : MessageType, Value : MessageValues]
```

Used to print variable values. Must be used with ∧ and in states which are reached by *TLC*.

```
PrintVal(message, expression) ≜ Print(⟨message, expression⟩, TRUE)
```

Used to print the values of all variables.

```
PrintAllVariableValues ≜
    ∧ PrintVal("Estimates at round: ", estimatesAtRound)
    ∧ PrintVal("Proposals at round: ", proposalsAtRound)
    ∧ PrintVal("Decisions: ", decisions)
    ∧ PrintVal("Rounds: ", rounds)
    ∧ PrintVal("Common Coin:", commonCoin)
```

A set of all nodes other than the specified node.

```
AllOtherNodes(node) ≜
    NODES \ {node}
```

Returns TRUE if a node sent the specified message during the specified round.

```
DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue) ≜
    LET messageRecord ≜ [Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ messageValue]
    IN  messageRecord ∈ msgsHistory
```

Returns TRUE if a node sent a message of the specified type with any value ("0", "1", or "?") during the specified round.

```
DidNodeSendMessageOfType(sendingNode, nodeRound, messageType) ≜
    ∃ msgValue ∈ MessageValues :
        LET messageRecord ≜ [Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ msgValue]
        IN  messageRecord ∈ msgsHistory
```

Returns the number of messages sent by nodes other than *checkingNode* of the specified message type.

```
HowManyMessagesOfType(checkingNode, nodeRound, messageType) ≜
    Cardinality({node ∈ AllOtherNodes(checkingNode) :
        DidNodeSendMessageOfType(node, nodeRound, messageType)})
```

Adds a new message to the *msgsHistory* variable.

```
AddToMessageHistory(sendingNode, nodeRound, messageType, messageValue) ≜
    ∧ ¬DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue)
    ∧ msgsHistory' =
          msgsHistory ∪ {[Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ messageValue]}
```

The number of node estimates of a given estimated value made by nodes other than the checking node in the same round.

```
HowManyEstimates(checkingNode, estimatedValue) ≜
    Cardinality({node ∈ NODES :
        ∧ node ≠ checkingNode
        ∧ estimatesAtRound[rounds[checkingNode]][node] = estimatedValue})
```

Sends a proposal message with a value that is in more than $(N + F)$ / 2 estimates.

```
SendProposalMessage(node) ≜
    ∃ estimateValue ∈ DecisionValues :
        ∧ HowManyEstimates(node, estimateValue) ∗ 2 > NumberOfNodes + NumberOfFaultyNodes
        ∧ proposalsAtRound' = [proposalsAtRound EXCEPT ![rounds[node]][node] = estimateValue]
        ∧ AddToMessageHistory(node, rounds[node], "proposal", estimateValue)
```

Check if there are more than $(N + F)$ / 2 estimates of the same value.

```
$CheckPhase1a(node) \triangleq$
    $\exists\, estimateValue \in DecisionValues :$
        $HowManyEstimates(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes$
```

Wait for $N - F$ nodes with estimate values other than "-1" in the same round.

```
$WaitForReports(waitingNode) \triangleq$
    $Cardinality(\{node \in NODES :$
        $\land\ node \neq waitingNode$
        $\land\ estimatesAtRound[rounds[waitingNode]][node] \in DecisionValues\}) \geq NumberOfNodes - NumberOfFaultyNodes$
```

Wait for $N - F$ nodes to send report messages with their estimate values.

```
$WaitForReportMessages(node) \triangleq$
    $HowManyMessagesOfType(node, rounds[node], \text{"report"}) \geq NumberOfNodes - NumberOfFaultyNodes$
```

The first phase of the algorithm. All nodes check whether there are more than $(N + F)$ / 2 estimates with the same value v. If such a value exists, then v is sent to all other nodes in a proposal message. v is not necessarily the same as the node's current estimate value. A node can send
a value different from its own estimate value to other nodes in a proposal message.
If such a value doesn't exist, the node sends a proposal message with a "?" value. A faulty node always sends a proposal message with the "?" value.

```
$Phase1(node) \triangleq$
    $\land\ node \in NODES$
    $\land\ proposalsAtRound[rounds[node]][node] \notin ProposalValues$
    $\land\ WaitForReports(node)$
    $\land$ IF $rounds[node] = 1$    Send initial estimate values within report messages.
        THEN $WaitForReportMessages(node)$
        ELSE TRUE
    $\land\ DidNodeSendMessageOfType(node, rounds[node], \text{"report"})$
    $\land$ IF $\land\ CheckPhase1a(node)$
           $\land\ node \notin FAULTY\_NODES$
        THEN $SendProposalMessage(node)$
        ELSE $\land\ proposalsAtRound' = [proposalsAtRound$ EXCEPT $![rounds[node]][node] = \text{"?"}]$
             $\land\ AddToMessageHistory(node, rounds[node], \text{"proposal"}, \text{"?"})$
    $\land$ UNCHANGED $\langle estimatesAtRound, decisions, rounds, commonCoin, estimateHistory\rangle$
```

The number of node proposals of a given proposal value made by nodes other than the checking node in the same round.

```
$HowManyProposals(checkingNode, proposalValue) \triangleq$
    $Cardinality(\{node \in NODES :$
        $\land\ node \neq checkingNode$
        $\land\ proposalsAtRound[rounds[checkingNode]][node] = proposalValue\})$
```

Returns a set of proposal values "0" or "1" that are proposed by more than $(N + F)$ / 2 nodes.

```
$AtLeastMajorityProposals(node) \triangleq$
    $\{estimateValue \in DecisionValues :$
        $HowManyProposals(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes\}$
```

Returns a set of proposal values "0" or "1" that are proposed by at least $F + 1$ nodes.

```
$AtLeastNF1Proposals(node) \triangleq$
    $\{estimateValue \in DecisionValues :$
        $HowManyProposals(node, estimateValue) \geq NumberOfFaultyNodes + 1\}$
```

Check for $F + 1$ or more proposal messages with the same value.

```
$CheckPhase2a(node) \triangleq$
    $AtLeastMajorityProposals(node) \neq \{\}$
```

Check for $F + 1$ or more proposal messages with the same value.

```
$CheckPhase2b(node) \triangleq$
    $AtLeastNF1Proposals(node) \neq \{\}$
```

Decide an estimate value that has been proposed by more than $(N + F)$ / 2 nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$DecidePhase2a(node) \triangleq$
    $\exists\, estimateValue \in AtLeastMajorityProposals(node) :$
        $decisions' = [decisions$ EXCEPT $![node] = estimateValue]$
```

Estimate a proposal value that has been proposed by at least $F + 1$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$EstimatePhase2b(node) \triangleq$
    $\exists\, estimateValue \in AtLeastNF1Proposals(node) :$
        $\land\ estimatesAtRound' = [estimatesAtRound\ \text{EXCEPT}\ ![rounds[node]][node] = estimateValue]$
        $\land\ estimateHistory' = [estimateHistory\ \text{EXCEPT}\ ![node] = estimateValue]$
```

Pick am estimate value from "0" or "1" for a node based on the common coin of the current round.

```
$EstimatePhase2c(node) \triangleq$
    $\land\ commonCoin[rounds[node]] \neq$ "-1"
    $\land\ estimatesAtRound' = [estimatesAtRound\ \text{EXCEPT}\ ![rounds[node]][node] = commonCoin[rounds[node]]]$
    $\land\ estimateHistory' = [estimateHistory\ \text{EXCEPT}\ ![node] = commonCoin[rounds[node]]]$
```

Wait for $N - F$ proposal messages with values other than "-1" in the same round.

```
$WaitForProposals(waitingNode) \triangleq$
    $Cardinality(\{node \in NODES :$
        $\land\ node \neq waitingNode$
        $\land\ proposalsAtRound[rounds[waitingNode]][node] \in ProposalValues\}) \geq NumberOfNodes - NumberOfFaultyNodes$
```

Second phase of the *BenOr* consensus algorithm. All nodes check whether there have been more than $(N + F)$ / 2 proposal messages with the same value "0" or "1". If there were that many messages, each node updates their decision and estimate values to match the value in the proposal messages.
Otherwise, if there are at least $F + 1$ received proposal messages with the same value value "0" or "1", the value is set as the new estimate for the receiving node.
In all other cases, nodes estimate a random value of "0" or "1" for the next round.

```
$Phase2(node) \triangleq$
    $\land\ node \in NODES$
    $\land\ WaitForProposals(node)$
    $\land\ DidNodeSendMessageOfType(node, rounds[node],$ "proposal"$)$
    $\land$ IF $CheckPhase2a(node) \land CheckPhase2b(node)$
        THEN $DecidePhase2a(node) \land EstimatePhase2b(node)$   If there are more than $(N + F)$ / 2 valid proposals . .
        ELSE IF $\neg CheckPhase2a(node) \land CheckPhase2b(node)$
                THEN $EstimatePhase2b(node) \land$ UNCHANGED $\langle decisions \rangle$   If there are at least $F + 1$ valid proposals.
                ELSE IF $\neg CheckPhase2a(node) \land \neg CheckPhase2b(node)$
                        THEN $EstimatePhase2c(node) \land$ UNCHANGED $\langle decisions \rangle$   Estimates are randomized.
                        ELSE FALSE   No other cases possible.
    $\land\ rounds' = [rounds\ \text{EXCEPT}\ ![node] = @ + 1]$
    $\land$ UNCHANGED $\langle proposalsAtRound, commonCoin, msgsHistory \rangle$
```

Set the node's next round's estimate to the previous round's estimate value so that other nodes are aware that the node has entered the next round. In the previous round the estimate will be some valid value other than "-1".

```
$SetNextRoundEstimate(node) \triangleq$
    $\land\ node \in NODES$
    $\land$ IF $rounds[node] = 1 \land estimatesAtRound[rounds[node]][node] \neq$ "-1"
        THEN $\land\ AddToMessageHistory(node, rounds[node],$ "report"$, estimatesAtRound[rounds[node]][node])$
             $\land$ UNCHANGED $\langle estimatesAtRound \rangle$
        ELSE $\land\ estimatesAtRound[rounds[node]][node] =$ "-1"
             $\land\ estimatesAtRound' =$
                  $[estimatesAtRound\ \text{EXCEPT}\ ![rounds[node]][node] = estimatesAtRound[rounds[node] - 1][node]]$
             $\land\ AddToMessageHistory(node, rounds[node],$ "report"$, estimatesAtRound[rounds[node] - 1][node])$
    $\land$ UNCHANGED $\langle proposalsAtRound, decisions, rounds, commonCoin, estimateHistory \rangle$
```

A node sets the Common Coin value for the current round the node is in. If in the round at least 1 node has already estimated some value non-randomly, then that round's common coin
value will be set to the node's estimate value (the Common Coin was lucky).
Otherwise, the Common Coin value is set to "0" or "1" and *TLC* will check behaviors when all nodes randomly estimate "0" or "1" together.

```
$TossCommonCoin(settingNode) \triangleq$
    $\land\ commonCoin[rounds[settingNode]] =$ "-1"
    $\land\ \lor \exists\, node \in NODES \setminus FAULTY\_NODES :$   If at least 1 node already estimated some value in phase $2b$.
            $\land\ estimatesAtRound[rounds[settingNode]][node] \neq$ "-1"
            $\land\ WaitForProposals(node)$
            $\land\ DidNodeSendMessageOfType(node, rounds[node],$ "proposal"$)$
            $\land\ \lor CheckPhase2a(node) \land CheckPhase2b(node)$
              $\lor \neg CheckPhase2a(node) \land CheckPhase2b(node)$
              ∧ commonCoin′ =
                  [commonCoin EXCEPT ![rounds[settingNode]] = estimatesAtRound[rounds[settingNode]][node]]
        ∨  ∧ ∀ node ∈ NODES \ FAULTY_NODES :
              ∧ WaitForProposals(node)
              ∧ DidNodeSendMessageOfType(node, rounds[node], "proposal")
              ∧ ¬CheckPhase2a(node)
              ∧ ¬CheckPhase2b(node)
           ∧ ∃ coinValue ∈ DecisionValues :   All nodes estimate both values based on the Common Coin.
              commonCoin′ = [commonCoin EXCEPT ![rounds[settingNode]] = coinValue]
     ∧ UNCHANGED ⟨estimatesAtRound, proposalsAtRound, decisions, rounds, msgsHistory, estimateHistory⟩
```

Consensus property for *TLC* to check. Consensus is guaranteed only between non-faulty nodes. All decision values must be "0" or "1" and all non-faulty nodes must decide
on the same value.

```
ConsensusReachedByzantine ≜
     ∧ ∀ node ∈ NODES \ FAULTY_NODES : decisions[node] ∈ DecisionValues
     ∧ ∀ node1, node2 ∈ NODES \ FAULTY_NODES : decisions[node1] = decisions[node2]
```

Common Coin validity property for *TLC* to check. The *CC* is valid, if its actually set during every round.

```
CommonCoinValid ≜
     ∧ ∀ round ∈ NodeRounds : commonCoin[round] ∈ DecisionValues
```

The last state of the algorithm when consensus is reached. Used to stop the algorithm by omitting behaviors after consensus is reached.

```
LastState ≜
     ∧ ConsensusReachedByzantine

TypeOK ≜
     ∧    ∀ round ∈ NodeRounds : estimatesAtRound[round] ∈ AllNodeEstimateFunctions
     ∧    ∀ round ∈ NodeRounds : proposalsAtRound[round] ∈ NodeProposalFunctions
     ∧    ∀ node ∈ NODES : decisions[node] ∈ NodeDecisionValues
     ∧    ∀ node ∈ NODES : rounds[node] ∈ NodeRounds
     ∧    ∀ round ∈ NodeRounds : commonCoin[round] ∈ CommonCoinValues
     ∧    msgsHistory ⊆ MessageRecordSet
```

If *CHECK_ALL_INITIAL_VALUES* is set to TRUE, *TLC* will check all possible estimate value combinations. Otherwise, initial estimates are defined by configuring what nodes estimate a "0" or a "1" value with the sets *ESTIMATE_ZERO* and *ESTIMATE_ONE*.
Estimates for all other rounds are set to "-1". Proposal message values for all rounds are set to "-1". Initial decisions are set to "-1" at the start to ensure *ConsensusReached* is not true
in the initial state.
Initially no Common Coins were tossed at any round yet.

```
Init ≜
     ∧ IF CHECK_ALL_INITIAL_VALUES
       THEN ∃ estimateFunction   ∈ ValidNodeEstimateFunctions :
              ∧ estimatesAtRound = [round ∈ NodeRounds ↦ IF round = 1
                                                         THEN estimateFunction
                                                         ELSE [node ∈ NODES ↦ "-1"]]
              ∧ estimateHistory = estimateFunction
       ELSE   ∧ estimatesAtRound = [round ∈ NodeRounds ↦ IF round = 1
                                                         THEN [node ∈ NODES ↦
                                                           IF node ∈ ESTIMATE_ZERO
                                                            THEN "0"
                                                            ELSE "1"]
                                                         ELSE [node ∈ NODES ↦ "-1"]]
              ∧ estimateHistory = [node ∈ NODES ↦ IF node ∈ ESTIMATE_ZERO THEN "0" ELSE "1"]
     ∧ proposalsAtRound = [round ∈ NodeRounds ↦ [node ∈ NODES ↦ "-1"]]
     ∧ decisions = [node ∈ NODES ↦ "-1"]
     ∧ rounds = [node ∈ NODES ↦ 1]
     ∧ commonCoin = [round ∈ NodeRounds ↦ "-1"]
     ∧ msgsHistory = {}

Next ≜
    ∃ node ∈ NODES :
       ∧ ¬LastState   Stops TLC when used with deadlock check enabled.
    ∧ ∨ $Phase1(node)$
      ∨ $Phase2(node)$
      ∨ $TossCommonCoin(node)$
      ∨ $SetNextRoundEstimate(node)$

$Spec \triangleq$
    ∧ $Init$
    ∧ $\Box[Next]_{vars}$
    ∧ $\forall\, node \in NODES$ :
        $\text{WF}_{vars}(Phase1(node) \lor Phase2(node) \lor TossCommonCoin(node) \lor SetNextRoundEstimate(node))$

VERY IMPORTANT: when checking the RefinementProperty, all INSTANCE statements in the other refined history variable specification
 .tla file must be commented out to avoid circular dependencies.
Also, the RefinementProperty definition must be commented out as well since the INSTANCE statements would not be defined in such a case.
$RefMsgsH \triangleq$
    INSTANCE $BenOrMsgsByzCCLH$
     WITH $msgs \leftarrow msgsHistory$,
          $estimates \leftarrow estimateHistory$,
          $estimateHistory \leftarrow estimatesAtRound$,
          $proposalHistory \leftarrow proposalsAtRound$

$RefMsgs \triangleq$
    INSTANCE $BenOrMsgsByzCCL$
     WITH $msgs \leftarrow msgsHistory$,
          $estimates \leftarrow estimateHistory$

$RefAbstCCL \triangleq$
    INSTANCE $BenOrAbstByzCCL$

$RefinementProperty \triangleq$    Can be uncommented here and in the BenOrByzCC.cfg file to check refinement.
    ∧ $RefMsgs!Spec$
    ∧ $RefMsgsH!Spec$
    ∧ $RefAbstCCL!Spec$
$ConsensusProperty \triangleq \Diamond\Box[ConsensusReachedByzantine]_{vars}$  Eventually, consensus will always be reached.
$CommonCoinProperty \triangleq \Diamond\Box[CommonCoinValid]_{vars}$  Eventually, the CC will be set for all rounds.
├──────────────────────────────────────────────────────────────────┤
THEOREM $Spec \Rightarrow \Box TypeOK$  Checked by TLC.
PROOF OMITTED
├──────────────────────────────────────────────────────────────────┤
THEOREM $Spec \Rightarrow ConsensusProperty$  Checked by TLC.
PROOF OMITTED
├──────────────────────────────────────────────────────────────────┤
THEOREM $Spec \Rightarrow CommonCoinProperty$  Checked by TLC.
PROOF OMITTED
└──────────────────────────────────────────────────────────────────┘
```

Figure 31. The BenOrAbstByzCCLH TLA^{+} specification

MODULE *BenOrAbstByzCCLRst*

This specification is a refinement of *BenOrAbstByzCCL* that allows for only one non-faulty node *CCNode* to toss the common coin. The issue is that if we want to model nodes failing or becoming faulty during the execution of the algorithm, then *CCNode* node must not be compromised or the algorithm won't be able to progress.

Other *BenOrAbstByz* common coin specifications have their own disadvantages : – In *BenOrAbstByzCC*, the coin is tossed by the system, no one specific node, yet the state space is a lot greater due to the
coin toss being a system action.

- In *BenOrAbstByzCCL*, any node can toss the coin, but this leads to a growth of the total number of states as the number of nodes increases.

The refinement mapping *RefAbstCCL* is also included in the specification and can be checked with *TLC*. The specification contains fewer distinct and total states compared to *BenOrAbstByzCCL* and as the number
of nodes increase the amount of total states remains lower than *BenOrAbstByzCCL*.

EXTENDS *TLC*, *Integers*, *FiniteSets*

Use the *BenOrByzCC.cfg* file.

CONSTANTS *NODES*, *FAULTY_NODES*, *ESTIMATE_ZERO*, *ESTIMATE_ONE*, *CHECK_ALL_INITIAL_VALUES*

VARIABLE *estimatesAtRound*	Function that returns a function of all the node's estimates at the specified round.
VARIABLE *proposalsAtRound*	Function that returns a function of all the node's proposal values at the specified round.
VARIABLE *decisions*	Function that returns a given node's current decision value.
VARIABLE *rounds*	Function that returns a given node's current round.
VARIABLE *commonCoin*	Models the Common Coin.

$vars \triangleq \langle estimatesAtRound, proposalsAtRound, decisions, rounds, commonCoin \rangle$

For the *BenOr Byzantine* consensus protocol version, $N > 5 * F$ must be true. Also, F must be greater than 0, otherwise the $N - F$ check for proposal messages won't work.

ASSUME $\land\ (Cardinality(NODES) > 5 * Cardinality(FAULTY\_NODES))$
$\qquad\quad \land\ (Cardinality(FAULTY\_NODES) \geq 1)$

Nodes and their decisions must be specified.

ASSUME $\land\ NODES \neq \{\}$
$\qquad\quad \land\ ESTIMATE\_ONE \neq \{\}$
$\qquad\quad \land\ ESTIMATE\_ZERO \neq \{\}$

Decision values must be specified for all nodes.

ASSUME $Cardinality(ESTIMATE\_ONE) + Cardinality(ESTIMATE\_ZERO) = Cardinality(NODES)$

Numbers of nodes and faulty nodes.

$NumberOfNodes \triangleq Cardinality(NODES)$
$NumberOfFaultyNodes \triangleq Cardinality(FAULTY\_NODES)$

A non-faulty node that sets the common coin.

$CCNode \triangleq$ CHOOSE $node \in NODES : node \notin FAULTY\_NODES$

Only a limited number of rounds is modeled.

$NodeRounds \triangleq 1 \ldots 3$

Decision values of binary consensus.

$DecisionValues \triangleq \{$"0", "1"$\}$

Set of all possible node decision values including the "-1" used for the initial state.

$NodeDecisionValues \triangleq DecisionValues \cup \{$"-1"$\}$

Set of possible proposal values.

$ProposalValues \triangleq \{$"?", "0", "1"$\}$

Set of all possible proposal message values including the "-1" used to indicate a proposal was not yet sent.

$ProposalMessageValues \triangleq ProposalValues \cup \{$"-1"$\}$

Set of all possible node estimate values. The "-1" is used to track whether a node has sent a report message or not.

$NodeEstimateValues \triangleq DecisionValues \cup \{$"-1"$\}$

Set of all functions that map nodes to their estimate values and "-1".

$AllNodeEstimateFunctions \triangleq [NODES \rightarrow NodeEstimateValues]$

Set of all functions that map nodes to their estimate values "0" or "1".

$ValidNodeEstimateFunctions \triangleq [NODES \rightarrow DecisionValues]$

Set of all functions that map nodes to their proposal message values.

```
NodeProposalFunctions ≜ [NODES → ProposalMessageValues]
```

Set of all possible Common Coin values. "-1" means that the common coin was not tossed yet for that round.

```
CommonCoinValues ≜ {"-1", "0", "1"}
```

Used to print variable values. Must be used with $\land$ and in states which are reached by *TLC*.

```
PrintVal(message, expression) ≜ Print(⟨message, expression⟩, TRUE)
```

Used to print the values of all variables.

```
PrintAllVariableValues ≜
    ∧ PrintVal("Estimates at round: ", estimatesAtRound)
    ∧ PrintVal("Proposals at round: ", proposalsAtRound)
    ∧ PrintVal("Decisions: ", decisions)
    ∧ PrintVal("Rounds: ", rounds)
    ∧ PrintVal("Common Coin:", commonCoin)
```

A set of all nodes other than the specified node.

```
AllOtherNodes(node) ≜
    NODES \ {node}
```

The number of node estimates of a given estimated value made by nodes other than the checking node in the same round.

```
HowManyEstimates(checkingNode, estimatedValue) ≜
    Cardinality({node ∈ NODES :
        ∧ node ≠ checkingNode
        ∧ estimatesAtRound[rounds[checkingNode]][node] = estimatedValue})
```

Sends a proposal message with a value that is in more than $(N + F)$ / 2 estimates.

```
SendProposalMessage(node) ≜
    ∃ estimateValue ∈ DecisionValues :
        ∧ HowManyEstimates(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes
        ∧ proposalsAtRound' = [proposalsAtRound EXCEPT ![rounds[node]][node] = estimateValue]
```

Check if there are more than $(N + F)$ / 2 estimates of the same value.

```
CheckPhase1a(node) ≜
    ∃ estimateValue ∈ DecisionValues :
        HowManyEstimates(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes
```

Wait for $N - F$ nodes with estimate values other than "-1" in the same round.

```
WaitForReports(waitingNode) ≜
    Cardinality({node ∈ NODES :
        ∧ node ≠ waitingNode
        ∧ estimatesAtRound[rounds[waitingNode]][node] ∈ DecisionValues}) ≥ NumberOfNodes − NumberOfFaultyNodes
```

The first phase of the algorithm. All nodes check whether there are more than $(N + F)$ / 2 estimates with the same value v. If such a value exists, then v is sent to all other nodes in a proposal message. v is not necessarily the same as the node's current estimate value. A node can send
a value different from its own estimate value to other nodes in a proposal message.
If such a value doesn't exist, the node sends a proposal message with a "?" value. A faulty node always sends a proposal message with the "?" value.

```
Phase1(node) ≜
    ∧ node ∈ NODES
    ∧ proposalsAtRound[rounds[node]][node] ∉ ProposalValues
    ∧ WaitForReports(node)
    ∧ IF ∧ CheckPhase1a(node)
         ∧ node ∉ FAULTY_NODES
      THEN SendProposalMessage(node)
      ELSE proposalsAtRound' = [proposalsAtRound EXCEPT ![rounds[node]][node] = "?"]
    ∧ UNCHANGED ⟨estimatesAtRound, decisions, rounds, commonCoin⟩
```

The number of node proposals of a given proposal value made by nodes other than the checking node in the same round.

```
HowManyProposals(checkingNode, proposalValue) ≜
    Cardinality({node ∈ NODES :
        ∧ node ≠ checkingNode
        ∧ proposalsAtRound[rounds[checkingNode]][node] = proposalValue})
```

Returns a set of proposal values “0” or “1” that are proposed by more than $(N + F)$ / 2 nodes.

```
AtLeastMajorityProposals(node) ≜
    {estimateValue ∈ DecisionValues :
        HowManyProposals(node, estimateValue) ∗ 2 > NumberOfNodes + NumberOfFaultyNodes}
```

Returns a set of proposal values “0” or “1” that are proposed by at least $F + 1$ nodes.

```
AtLeastNF1Proposals(node) ≜
    {estimateValue ∈ DecisionValues :
        HowManyProposals(node, estimateValue) ≥ NumberOfFaultyNodes + 1}
```

Check for $F + 1$ or more proposal messages with the same value.

```
CheckPhase2a(node) ≜
    AtLeastMajorityProposals(node) ≠ {}
```

Check for $F + 1$ or more proposal messages with the same value.

```
CheckPhase2b(node) ≜
    AtLeastNF1Proposals(node) ≠ {}
```

Decide an estimate value that has been proposed by more than $(N + F)$ / 2 nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
DecidePhase2a(node) ≜
    ∃ estimateValue ∈ AtLeastMajorityProposals(node) :
        decisions′ = [decisions EXCEPT ![node] = estimateValue]
```

Estimate a proposal value that has been proposed by at least $F + 1$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
EstimatePhase2b(node) ≜
    ∃ estimateValue ∈ AtLeastNF1Proposals(node) :
        estimatesAtRound′ = [estimatesAtRound EXCEPT ![rounds[node]][node] = estimateValue]
```

Pick an estimate value from “0” or “1” for a node based on the common coin of the current round.

```
EstimatePhase2c(node) ≜
    ∧ commonCoin[rounds[node]] ≠ “-1”
    ∧ estimatesAtRound′ = [estimatesAtRound EXCEPT ![rounds[node]][node] = commonCoin[rounds[node]]]
```

Wait for $N - F$ proposal messages with values other than “-1” in the same round.

```
WaitForProposals(waitingNode) ≜
    Cardinality({node ∈ NODES :
        ∧ node ≠ waitingNode
        ∧ proposalsAtRound[rounds[waitingNode]][node] ∈ ProposalValues}) ≥ NumberOfNodes − NumberOfFaultyNodes
```

Second phase of the *BenOr* consensus algorithm. All nodes check whether there have been more than $(N + F)$ / 2 proposal messages with the same value “0” or “1”. If there were that many messages, each node updates their decision and estimate values to match the value in the proposal messages.
Otherwise, if there are at least $F + 1$ received proposal messages with the same value value “0” or “1”, the value is set as the new estimate for the receiving node.
In all other cases, nodes estimate a random value of “0” or “1” for the next round.

```
Phase2(node) ≜
    ∧ node ∈ NODES
    ∧ WaitForProposals(node)
    ∧ IF CheckPhase2a(node) ∧ CheckPhase2b(node)
        THEN DecidePhase2a(node) ∧ EstimatePhase2b(node)   If there are more than (N + F) / 2 valid proposals.
        ELSE IF ¬CheckPhase2a(node) ∧ CheckPhase2b(node)
                THEN EstimatePhase2b(node) ∧ UNCHANGED ⟨decisions⟩   If there are at least F + 1 valid proposals.
                ELSE IF ¬CheckPhase2a(node) ∧ ¬CheckPhase2b(node)
                        THEN EstimatePhase2c(node) ∧ UNCHANGED ⟨decisions⟩   Estimates are randomized.
                        ELSE FALSE   No other cases possible.
    ∧ rounds′ = [rounds EXCEPT ![node] = @ + 1]
    ∧ UNCHANGED ⟨proposalsAtRound, commonCoin⟩
```

Set the node’s next round’s estimate to the previous round’s estimate value so that other nodes are aware that the node has entered the next round. In the previous round the estimate will be some valid value other than “-1”.

```
SetNextRoundEstimate(node) ≜
    ∧ node ∈ NODES
    ∧ estimatesAtRound[rounds[node]][node] = “-1”
    $\land\ estimatesAtRound' = [estimatesAtRound\ \text{EXCEPT}\ ![rounds[node]][node] = estimatesAtRound[rounds[node] - 1][node]]$
    $\land\ \text{UNCHANGED}\ \langle proposalsAtRound,\ decisions,\ rounds,\ commonCoin \rangle$
```

CCNode sets the Common Coin value for the current round the node is in. If in the round at least 1 node has already estimated some value non-randomly, then that round's common coin
value will be set to the node's estimate value (the Common Coin was lucky).
Otherwise, the Common Coin value is set to "0" or "1" and *TLC* will check behaviors when all nodes randomly estimate "0" or "1" together.

```
$TossCommonCoin(settingNode) \triangleq$
    $\land\ settingNode = CCNode$
    $\land\ commonCoin[rounds[settingNode]] = \text{"-1"}$
    $\land\ \lor\ \exists\, node \in NODES \setminus FAULTY\_NODES :$    If at least 1 node already estimated some value in phase $2b$.
            $\land\ estimatesAtRound[rounds[settingNode]][node] \neq \text{"-1"}$
            $\land\ WaitForProposals(node)$
            $\land\ \lor\ CheckPhase2a(node) \land CheckPhase2b(node)$
               $\lor\ \neg CheckPhase2a(node) \land CheckPhase2b(node)$
            $\land\ commonCoin' =$
                $[commonCoin\ \text{EXCEPT}\ ![rounds[settingNode]] = estimatesAtRound[rounds[settingNode]][node]]$
      $\lor\ \land\ \forall\, node \in NODES \setminus FAULTY\_NODES :$
            $\land\ WaitForProposals(node)$
            $\land\ \neg CheckPhase2a(node)$
            $\land\ \neg CheckPhase2b(node)$
          $\land\ \exists\, coinValue \in DecisionValues :$    All nodes estimate both values based on the Common Coin.
            $commonCoin' = [commonCoin\ \text{EXCEPT}\ ![rounds[settingNode]] = coinValue]$
    $\land\ \text{UNCHANGED}\ \langle estimatesAtRound,\ proposalsAtRound,\ decisions,\ rounds \rangle$
```

Consensus property for *TLC* to check. Consensus is guaranteed only between non-faulty nodes. All decision values must be "0" or "1" and all non-faulty nodes must decide
on the same value.

```
$ConsensusReachedByzantine \triangleq$
    $\land\ \forall\, node \in NODES \setminus FAULTY\_NODES : decisions[node] \in DecisionValues$
    $\land\ \forall\, node1,\ node2 \in NODES \setminus FAULTY\_NODES : decisions[node1] = decisions[node2]$
```

Common Coin validity property for *TLC* to check. The *CC* is valid, if its actually set during every round.

```
$CommonCoinValid \triangleq$
    $\land\ \forall\, round \in NodeRounds : commonCoin[round] \in DecisionValues$
```

The last state of the algorithm when consensus is reached. Used to stop the algorithm by omitting behaviors after consensus is reached.

```
$LastState \triangleq$
    $\land\ ConsensusReachedByzantine$

$TypeOK \triangleq$
    $\land\ \ \forall\, round \in NodeRounds : estimatesAtRound[round] \in AllNodeEstimateFunctions$
    $\land\ \ \forall\, round \in NodeRounds : proposalsAtRound[round] \in NodeProposalFunctions$
    $\land\ \ \forall\, node \in NODES : decisions[node] \in NodeDecisionValues$
    $\land\ \ \forall\, node \in NODES : rounds[node] \in NodeRounds$
    $\land\ \ \forall\, round \in NodeRounds : commonCoin[round] \in CommonCoinValues$
```

If *CHECK_ALL_INITIAL_VALUES* is set to TRUE, *TLC* will check all possible estimate value combinations. Otherwise, initial estimates
are defined by configuring what nodes estimate a "0" or a "1" value with the sets *ESTIMATE_ZERO* and *ESTIMATE_ONE*.
Estimates for all other rounds are set to "-1". Proposal message values for all rounds are set to "-1". Initial decisions are set to "-1" at the start to ensure *ConsensusReached* is not true
in the initial state.
Initially no Common Coins were tossed at any round yet.

```
$Init \triangleq$
    $\land\ \text{IF}\ CHECK\_ALL\_INITIAL\_VALUES$
        $\text{THEN}\ \exists\, estimateFunction \in ValidNodeEstimateFunctions :$
            $estimatesAtRound = [round \in NodeRounds \mapsto \text{IF}\ round = 1$
                                                       $\text{THEN}\ estimateFunction$
                                                       $\text{ELSE}\ [node \in NODES \mapsto \text{"-1"}]]$
        $\text{ELSE}\ \ estimatesAtRound = [round \in NodeRounds \mapsto \text{IF}\ round = 1$
                                                       $\text{THEN}\ [node \in NODES \mapsto$
                                                          $\text{IF}\ node \in ESTIMATE\_ZERO$
                                                           $\text{THEN}\ \text{"0"}$
                                                           $\text{ELSE}\ \ \text{"1"}]$
```

```
                                            ELSE  $[node \in NODES \mapsto \text{"-1"}]]$
    $\land$ $proposalsAtRound = [round \in NodeRounds \mapsto [node \in NODES \mapsto \text{"-1"}]]$
    $\land$ $decisions = [node \in NODES \mapsto \text{"-1"}]$
    $\land$ $rounds = [node \in NODES \mapsto 1]$
    $\land$ $commonCoin = [round \in NodeRounds \mapsto \text{"-1"}]$

$Next \triangleq$
    $\exists\, node \in NODES :$
        $\land \neg LastState$  Stops TLC when used with deadlock check enabled.
        $\land$ $\lor Phase1(node)$
          $\lor Phase2(node)$
          $\lor TossCommonCoin(node)$
          $\lor SetNextRoundEstimate(node)$

$Spec \triangleq$
    $\land Init$
    $\land \Box[Next]_{vars}$
    $\land \forall\, node \in NODES :$
        $\mathrm{WF}_{vars}(Phase1(node) \lor Phase2(node) \lor TossCommonCoin(node) \lor SetNextRoundEstimate(node))$

$RefAbstCCL \triangleq$
    INSTANCE $BenOrAbstByzCCL$

 RefinementProperty ≜ RefAbstCCL!Spec \* Can be uncommented here and in the BenOrByzCC.cfg file to check refinement.
$ConsensusProperty \triangleq \Diamond\Box[ConsensusReachedByzantine]_{vars}$  Eventually, consensus will always be reached.
$CommonCoinProperty \triangleq \Diamond\Box[CommonCoinValid]_{vars}$  Eventually, the CC will be set for all rounds.
---------------------------------------------------------------------------------------
THEOREM $Spec \Rightarrow \Box TypeOK$  Checked by TLC.
PROOF OMITTED
---------------------------------------------------------------------------------------
THEOREM $Spec \Rightarrow ConsensusProperty$  Checked by TLC.
PROOF OMITTED
---------------------------------------------------------------------------------------
THEOREM $Spec \Rightarrow CommonCoinProperty$  Checked by TLC.
PROOF OMITTED
=======================================================================================
```

Figure 32. The BenOrAbstByzCCLRst TLA$^+$ specification

MODULE *BenOrAbstByzCCLRstH*

This is a refinement of *BenOrMsgsByzCCLRst*, *BenOrAbstByzCCLRst*, and *BenOrMsgsByzCCLRstH* with additional refinement history variables. Several modifications were required for *TLC* to successfully verify the refinement mappings *RefMsgs* :

− Added the *msgsHistory* and *estimateHistory* variables that correspond to the *msgs* and estimates variables from *BenOrMsgsByzCCLRst*.

- Modified the specification so that proposal messages are added to the *msgsHistory* variable when they are sent. Also, faulty node proposals with "?" values are added to *msgsHistory* too. Likewise, report messages with estimate values are added in the action formula *SetNextRoundEstimate*.

- Made it so that whenever a node estimates a new value, then *estimateHistory* is also changed.

- Modified the *Init* formula to also set the *estimateHistory* values based on the initial estimates similarly to *BenOrMsgsByzCCLRst*.

- Added actions for checking whether certain messages are present in the *msgsHistory* variable. Added corresponding checks for whether specific messages were sent by a node in phase 1, phase 2, and the *TossCommonCoin* action. This does not alter the state space in a way to make the mapping *RefAbstCCLRst* incorrect, and is required for *TLC* to verify the mapping *RefMsgs*.

- Made it so that during the first phase of the first round, at least $N - F$ initial estimates are sent as report messages before continuing to the second phase.

The refinement mappings *RefMsgsH* and *RefAbstCCLRst* are also provided and can be checked with *TLC* if put inside the *RefinementProperty* definition before '!*Spec*'.

EXTENDS *TLC*, *Integers*, *FiniteSets*

Use the *BenOrByzCC.cfg* file.

CONSTANTS *NODES*, *FAULTY_NODES*, *ESTIMATE_ZERO*, *ESTIMATE_ONE*, *CHECK_ALL_INITIAL_VALUES*

VARIABLE *estimatesAtRound*	Function that returns a function of all the node's estimates at the specified round.
VARIABLE *proposalsAtRound*	Function that returns a function of all the node's proposal values at the specified round.
VARIABLE *decisions*	Function that returns a given node's current decision value.
VARIABLE *rounds*	Function that returns a given node's current round.
VARIABLE *commonCoin*	Models the Common Coin.
VARIABLE *msgsHistory*	History variables for refinement.
VARIABLE *estimateHistory*	

$vars \triangleq \langle estimatesAtRound, proposalsAtRound, decisions, rounds, commonCoin, msgsHistory, estimateHistory \rangle$

For the *BenOr Byzantine* consensus protocol version, $N > 5 * F$ must be true. Also, F must be greater than 0, otherwise the $N - F$ check for proposal messages won't work.

ASSUME $\land (Cardinality(NODES) > 5 * Cardinality(FAULTY\_NODES))$
$\land (Cardinality(FAULTY\_NODES) \geq 1)$

Nodes and their decisions must be specified.

ASSUME $\land NODES \neq \{\}$
$\land ESTIMATE\_ONE \neq \{\}$
$\land ESTIMATE\_ZERO \neq \{\}$

Decision values must be specified for all nodes.

ASSUME $Cardinality(ESTIMATE\_ONE) + Cardinality(ESTIMATE\_ZERO) = Cardinality(NODES)$

Numbers of nodes and faulty nodes.

$NumberOfNodes \triangleq Cardinality(NODES)$
$NumberOfFaultyNodes \triangleq Cardinality(FAULTY\_NODES)$

A non-faulty node that sets the common coin.

$CCNode \triangleq$ CHOOSE $node \in NODES : node \notin FAULTY\_NODES$

Only a limited number of rounds is modeled.

$NodeRounds \triangleq 1 \,..\, 3$

Decision values of binary consensus.

$DecisionValues \triangleq \{$"0", "1"$\}$

Set of all possible node decision values including the "-1" used for the initial state.

$NodeDecisionValues \triangleq DecisionValues \cup \{$"-1"$\}$

Set of possible proposal values.

$ProposalValues \triangleq \{$"?", "0", "1"$\}$

Set of all possible proposal message values including the "-1" used to indicate a proposal was not yet sent.

$ProposalMessageValues \triangleq ProposalValues \cup \{$"-1"$\}$

Set of all possible node estimate values. The "-1" is used to track whether a node has sent a report message or not.

```
NodeEstimateValues ≜ DecisionValues ∪ {"-1"}

 Set of all functions that map nodes to their estimate values and "-1".
AllNodeEstimateFunctions ≜ [NODES → NodeEstimateValues]

 Set of all functions that map nodes to their estimate values "0" or "1".
ValidNodeEstimateFunctions ≜ [NODES → DecisionValues]

 Set of all functions that map nodes to their proposal message values.
NodeProposalFunctions ≜ [NODES → ProposalMessageValues]

 Set of all possible Common Coin values. "-1" means that the common coin was not tossed yet for that round.
CommonCoinValues ≜ {"-1", "0", "1"}

 Possible values and types of messages in the BenOr consensus algorithm.
MessageType ≜ {"report", "proposal"}
MessageValues ≜ {"0", "1", "?"}

 Each message record also has a round number.
MessageRecordSet ≜ [Sender : NODES, Round : Nat, Type : MessageType, Value : MessageValues]

 Used to print variable values. Must be used with ∧ and in states which are reached by TLC.
PrintVal(message, expression) ≜ Print(⟨message, expression⟩, TRUE)

 Used to print the values of all variables.
PrintAllVariableValues ≜
    ∧ PrintVal("Estimates at round: ", estimatesAtRound)
    ∧ PrintVal("Proposals at round: ", proposalsAtRound)
    ∧ PrintVal("Decisions: ", decisions)
    ∧ PrintVal("Rounds: ", rounds)
    ∧ PrintVal("Common Coin:", commonCoin)

 A set of all nodes other than the specified node.
AllOtherNodes(node) ≜
    NODES \ {node}

 Returns TRUE if a node sent the specified message during the specified round.
DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue) ≜
    LET messageRecord ≜ [Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ messageValue]
    IN  messageRecord ∈ msgsHistory

 Returns TRUE if a node sent a message of the specified type with any value ("0", "1", or "?") during the specified round.
DidNodeSendMessageOfType(sendingNode, nodeRound, messageType) ≜
   ∃ msgValue ∈ MessageValues :
       LET messageRecord ≜ [Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ msgValue]
       IN  messageRecord ∈ msgsHistory

 Returns the number of messages sent by nodes other than checkingNode of the specified message type.
HowManyMessagesOfType(checkingNode, nodeRound, messageType) ≜
    Cardinality({node ∈ AllOtherNodes(checkingNode) :
        DidNodeSendMessageOfType(node, nodeRound, messageType)})

 Adds a new message to the msgsHistory variable.
AddToMessageHistory(sendingNode, nodeRound, messageType, messageValue) ≜
    ∧ ¬DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue)
    ∧ msgsHistory' =
        msgsHistory ∪ {[Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ messageValue]}

 The number of node estimates of a given estimated value made by nodes other than the checking node in the same round.
HowManyEstimates(checkingNode, estimatedValue) ≜
    Cardinality({node ∈ NODES :
        ∧ node ≠ checkingNode
        ∧ estimatesAtRound[rounds[checkingNode]][node] = estimatedValue})

 Sends a proposal message with a value that is in more than (N + F) / 2 estimates.
SendProposalMessage(node) ≜
```

```
    $\exists$ $estimateValue \in DecisionValues$ :
        $\land$ $HowManyEstimates(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes$
        $\land$ $proposalsAtRound' = [proposalsAtRound$ EXCEPT $![rounds[node]][node] = estimateValue]$
        $\land$ $AddToMessageHistory(node, rounds[node],$ "proposal", $estimateValue)$
```

Check if there are more than $(N + F)$ / 2 estimates of the same value.

```
$CheckPhase1a(node) \triangleq$
    $\exists$ $estimateValue \in DecisionValues$ :
        $HowManyEstimates(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes$
```

Wait for $N - F$ nodes with estimate values other than "-1" in the same round.

```
$WaitForReports(waitingNode) \triangleq$
    $Cardinality(\{node \in NODES$ :
        $\land$ $node \neq waitingNode$
        $\land$ $estimatesAtRound[rounds[waitingNode]][node] \in DecisionValues\}) \geq NumberOfNodes - NumberOfFaultyNodes$
```

Wait for $N - F$ nodes to send report messages with their estimate values.

```
$WaitForReportMessages(node) \triangleq$
    $HowManyMessagesOfType(node, rounds[node],$ "report"$) \geq NumberOfNodes - NumberOfFaultyNodes$
```

The first phase of the algorithm. All nodes check whether there are more than $(N + F)$ / 2 estimates with the same value v. If such a value exists, then v is sent to all other nodes in a proposal message. v is not necessarily the same as the node's current estimate value. A node can send
a value different from its own estimate value to other nodes in a proposal message.
If such a value doesn't exist, the node sends a proposal message with a "?" value. A faulty node always sends a proposal message with the "?" value.

```
$Phase1(node) \triangleq$
    $\land$ $node \in NODES$
    $\land$ $proposalsAtRound[rounds[node]][node] \notin ProposalValues$
    $\land$ $WaitForReports(node)$
    $\land$ IF $rounds[node] = 1$   Send initial estimate values within report messages.
        THEN $WaitForReportMessages(node)$
        ELSE TRUE
    $\land$ $DidNodeSendMessageOfType(node, rounds[node],$ "report"$)$
    $\land$ IF $\land$ $CheckPhase1a(node)$
           $\land$ $node \notin FAULTY\_NODES$
        THEN $SendProposalMessage(node)$
        ELSE $\land$ $proposalsAtRound' = [proposalsAtRound$ EXCEPT $![rounds[node]][node] =$ "?"$]$
             $\land$ $AddToMessageHistory(node, rounds[node],$ "proposal", "?"$)$
    $\land$ UNCHANGED $\langle estimatesAtRound, decisions, rounds, commonCoin, estimateHistory\rangle$
```

The number of node proposals of a given proposal value made by nodes other than the checking node in the same round.

```
$HowManyProposals(checkingNode, proposalValue) \triangleq$
    $Cardinality(\{node \in NODES$ :
        $\land$ $node \neq checkingNode$
        $\land$ $proposalsAtRound[rounds[checkingNode]][node] = proposalValue\})$
```

Returns a set of proposal values "0" or "1" that are proposed by more than $(N + F)$ / 2 nodes.

```
$AtLeastMajorityProposals(node) \triangleq$
    $\{estimateValue \in DecisionValues$ :
        $HowManyProposals(node, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes\}$
```

Returns a set of proposal values "0" or "1" that are proposed by at least $F + 1$ nodes.

```
$AtLeastNF1Proposals(node) \triangleq$
    $\{estimateValue \in DecisionValues$ :
        $HowManyProposals(node, estimateValue) \geq NumberOfFaultyNodes + 1\}$
```

Check for $F + 1$ or more proposal messages with the same value.

```
$CheckPhase2a(node) \triangleq$
    $AtLeastMajorityProposals(node) \neq \{\}$
```

Check for $F + 1$ or more proposal messages with the same value.

```
$CheckPhase2b(node) \triangleq$
    $AtLeastNF1Proposals(node) \neq \{\}$
```

Decide an estimate value that has been proposed by more than $(N + F)$ / 2 nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$DecidePhase2a(node) \triangleq$
    $\exists\, estimateValue \in AtLeastMajorityProposals(node) :$
        $decisions' = [decisions \text{ EXCEPT } ![node] = estimateValue]$
```

Estimate a proposal value that has been proposed by at least $F + 1$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$EstimatePhase2b(node) \triangleq$
    $\exists\, estimateValue \in AtLeastNF1Proposals(node) :$
        $\land\ estimatesAtRound' = [estimatesAtRound \text{ EXCEPT } ![rounds[node]][node] = estimateValue]$
        $\land\ estimateHistory' = [estimateHistory \text{ EXCEPT } ![node] = estimateValue]$
```

Pick am estimate value from “0” or “1” for a node based on the common coin of the current round.

```
$EstimatePhase2c(node) \triangleq$
    $\land\ commonCoin[rounds[node]] \neq$ “-1”
    $\land\ estimatesAtRound' = [estimatesAtRound \text{ EXCEPT } ![rounds[node]][node] = commonCoin[rounds[node]]]$
    $\land\ estimateHistory' = [estimateHistory \text{ EXCEPT } ![node] = commonCoin[rounds[node]]]$
```

Wait for $N - F$ proposal messages with values other than “-1” in the same round.

```
$WaitForProposals(waitingNode) \triangleq$
    $Cardinality(\{node \in NODES :$
        $\land\ node \neq waitingNode$
        $\land\ proposalsAtRound[rounds[waitingNode]][node] \in ProposalValues\}) \geq NumberOfNodes - NumberOfFaultyNodes$
```

Second phase of the *BenOr* consensus algorithm. All nodes check whether there have been more than $(N + F)$ / 2 proposal messages with the same value “0” or “1”. If there were that many messages, each node updates their decision and estimate values to match the value in the proposal messages.
Otherwise, if there are at least $F + 1$ received proposal messages with the same value value “0” or “1”, the value is set as the new estimate for the receiving node.
In all other cases, nodes estimate a random value of “0” or “1” for the next round.

```
$Phase2(node) \triangleq$
    $\land\ node \in NODES$
    $\land\ WaitForProposals(node)$
    $\land\ DidNodeSendMessageOfType(node, rounds[node],$ “proposal”$)$
    $\land$ IF $CheckPhase2a(node) \land CheckPhase2b(node)$
        THEN $DecidePhase2a(node) \land EstimatePhase2b(node)$    If there are more than $(N + F)$ / 2 valid proposals . .
        ELSE IF $\neg CheckPhase2a(node) \land CheckPhase2b(node)$
            THEN $EstimatePhase2b(node) \land$ UNCHANGED $\langle decisions \rangle$    If there are at least $F + 1$ valid proposals.
            ELSE IF $\neg CheckPhase2a(node) \land \neg CheckPhase2b(node)$
                THEN $EstimatePhase2c(node) \land$ UNCHANGED $\langle decisions \rangle$    Estimates are randomized.
                ELSE FALSE    No other cases possible.
    $\land\ rounds' = [rounds \text{ EXCEPT } ![node] = @ + 1]$
    $\land$ UNCHANGED $\langle proposalsAtRound, commonCoin, msgsHistory \rangle$
```

Set the node’s next round’s estimate to the previous round’s estimate value so that other nodes are aware that the node has entered the next round. In the previous round the estimate will be some valid value other than “-1”.

```
$SetNextRoundEstimate(node) \triangleq$
    $\land\ node \in NODES$
    $\land$ IF $rounds[node] = 1 \land estimatesAtRound[rounds[node]][node] \neq$ “-1”
        THEN $\land\ AddToMessageHistory(node, rounds[node],$ “report”$, estimatesAtRound[rounds[node]][node])$
             $\land$ UNCHANGED $\langle estimatesAtRound \rangle$
        ELSE $\land\ estimatesAtRound[rounds[node]][node] =$ “-1”
             $\land\ estimatesAtRound' =$
                  $[estimatesAtRound \text{ EXCEPT } ![rounds[node]][node] = estimatesAtRound[rounds[node] - 1][node]]$
             $\land\ AddToMessageHistory(node, rounds[node],$ “report”$, estimatesAtRound[rounds[node] - 1][node])$
    $\land$ UNCHANGED $\langle proposalsAtRound, decisions, rounds, commonCoin, estimateHistory \rangle$
```

CCNode sets the Common Coin value for the current round the node is in. If in the round at least 1 node has already estimated some value non-randomly, then that round’s common coin
value will be set to the node’s estimate value (the Common Coin was lucky).
Otherwise, the Common Coin value is set to “0” or “1” and *TLC* will check behaviors when all nodes randomly estimate “0” or “1” together.

```
$TossCommonCoin(settingNode) \triangleq$
    $\land\ settingNode = CCNode$
    $\land\ commonCoin[rounds[settingNode]] = \text{“-1”}$
    $\land\ \lor\ \exists\, node \in NODES \setminus FAULTY\_NODES :$   If at least 1 node already estimated some value in phase $2b$.
            $\land\ estimatesAtRound[rounds[settingNode]][node] \neq \text{“-1”}$
            $\land\ WaitForProposals(node)$
            $\land\ DidNodeSendMessageOfType(node, rounds[node], \text{“proposal”})$
            $\land\ \lor\ CheckPhase2a(node) \land CheckPhase2b(node)$
               $\lor\ \neg CheckPhase2a(node) \land CheckPhase2b(node)$
            $\land\ commonCoin' =$
               $[commonCoin\ \text{EXCEPT}\ ![rounds[settingNode]] = estimatesAtRound[rounds[settingNode]][node]]$
      $\lor\ \land\ \forall\, node \in NODES \setminus FAULTY\_NODES :$
            $\land\ WaitForProposals(node)$
            $\land\ DidNodeSendMessageOfType(node, rounds[node], \text{“proposal”})$
            $\land\ \neg CheckPhase2a(node)$
            $\land\ \neg CheckPhase2b(node)$
         $\land\ \exists\, coinValue \in DecisionValues :$   All nodes estimate both values based on the Common Coin.
            $commonCoin' = [commonCoin\ \text{EXCEPT}\ ![rounds[settingNode]] = coinValue]$
    $\land\ \text{UNCHANGED}\ \langle estimatesAtRound, proposalsAtRound, decisions, rounds, msgsHistory, estimateHistory\rangle$
```

Consensus property for *TLC* to check. Consensus is guaranteed only between non-faulty nodes. All decision values must be “0” or “1” and all non-faulty nodes must decide
on the same value.

```
$ConsensusReachedByzantine \triangleq$
    $\land\ \forall\, node \in NODES \setminus FAULTY\_NODES : decisions[node] \in DecisionValues$
    $\land\ \forall\, node1, node2 \in NODES \setminus FAULTY\_NODES : decisions[node1] = decisions[node2]$
```

Common Coin validity property for *TLC* to check. The *CC* is valid, if its actually set during every round.

```
$CommonCoinValid \triangleq$
    $\land\ \forall\, round \in NodeRounds : commonCoin[round] \in DecisionValues$
```

The last state of the algorithm when consensus is reached. Used to stop the algorithm by omitting behaviors after consensus is reached.

```
$LastState \triangleq$
    $\land\ ConsensusReachedByzantine$

$TypeOK \triangleq$
    $\land\ \ \forall\, round \in NodeRounds : estimatesAtRound[round] \in AllNodeEstimateFunctions$
    $\land\ \ \forall\, round \in NodeRounds : proposalsAtRound[round] \in NodeProposalFunctions$
    $\land\ \ \forall\, node \in NODES : decisions[node] \in NodeDecisionValues$
    $\land\ \ \forall\, node \in NODES : rounds[node] \in NodeRounds$
    $\land\ \ \forall\, round \in NodeRounds : commonCoin[round] \in CommonCoinValues$
    $\land\ \ msgsHistory \subseteq MessageRecordSet$
```

If *CHECK_ALL_INITIAL_VALUES* is set to TRUE, *TLC* will check all possible estimate value combinations. Otherwise, initial estimates are defined by configuring what nodes estimate a “0” or a “1” value with the sets *ESTIMATE_ZERO* and *ESTIMATE_ONE*.
Estimates for all other rounds are set to “-1”. Proposal message values for all rounds are set to “-1”. Initial decisions are set to “-1” at the start to ensure *ConsensusReached* is not true
in the initial state.
Initially no Common Coins were tossed at any round yet.

```
$Init \triangleq$
    $\land\ \text{IF}\ CHECK\_ALL\_INITIAL\_VALUES$
       $\text{THEN}\ \exists\, estimateFunction \in ValidNodeEstimateFunctions :$
            $\land\ estimatesAtRound = [round \in NodeRounds \mapsto \text{IF}\ round = 1$
                                                            $\text{THEN}\ estimateFunction$
                                                            $\text{ELSE}\ [node \in NODES \mapsto \text{“-1”}]]$
            $\land\ estimateHistory = estimateFunction$
       $\text{ELSE}\ \land\ estimatesAtRound = [round \in NodeRounds \mapsto \text{IF}\ round = 1$
                                                            $\text{THEN}\ [node \in NODES \mapsto$
                                                               $\text{IF}\ node \in ESTIMATE\_ZERO$
                                                                $\text{THEN}\ \text{“0”}$
                                                                $\text{ELSE}\ \text{“1”}]$
                                                            $\text{ELSE}\ [node \in NODES \mapsto \text{“-1”}]]$
            $\land\ estimateHistory = [node \in NODES \mapsto \text{IF}\ node \in ESTIMATE\_ZERO\ \text{THEN}\ \text{“0”}\ \text{ELSE}\ \text{“1”}]$
    ∧ proposalsAtRound = [round ∈ NodeRounds ↦ [node ∈ NODES ↦ "-1"]]
    ∧ decisions = [node ∈ NODES ↦ "-1"]
    ∧ rounds = [node ∈ NODES ↦ 1]
    ∧ commonCoin = [round ∈ NodeRounds ↦ "-1"]
    ∧ msgsHistory = {}

Next ≜
   ∃ node ∈ NODES :
      ∧ ¬LastState   Stops TLC when used with deadlock check enabled.
      ∧ ∨ Phase1(node)
        ∨ Phase2(node)
        ∨ TossCommonCoin(node)
        ∨ SetNextRoundEstimate(node)

Spec ≜
    ∧ Init
    ∧ □[Next]_vars
    ∧ ∀ node ∈ NODES :
        WF_vars(Phase1(node) ∨ Phase2(node) ∨ TossCommonCoin(node) ∨ SetNextRoundEstimate(node))

  VERY IMPORTANT: when checking the RefinementProperty, all INSTANCE statements in the other refined history variable specification
  .tla file must be commented out to avoid circular dependencies.
  Also, the RefinementProperty definition must be commented out as well since the INSTANCE statements would not be defined in such a case.
RefMsgsH ≜
   INSTANCE BenOrMsgsByzCCLRstH
    WITH msgs ← msgsHistory,
         estimates ← estimateHistory,
         estimateHistory ← estimatesAtRound,
         proposalHistory ← proposalsAtRound

RefMsgs ≜
   INSTANCE BenOrMsgsByzCCLRst
    WITH msgs ← msgsHistory,
         estimates ← estimateHistory

RefAbstCCLRst ≜
   INSTANCE BenOrAbstByzCCLRst

RefinementProperty ≜     Can be uncommented here and in the BenOrByzCC.cfg file to check refinement.
    ∧ RefMsgs!Spec
    ∧ RefMsgsH!Spec
    ∧ RefAbstCCLRst!Spec
ConsensusProperty ≜ ◇□[ConsensusReachedByzantine]_vars   Eventually, consensus will always be reached.
CommonCoinProperty ≜ ◇□[CommonCoinValid]_vars   Eventually, the CC will be set for all rounds.
├──────────────────────────────────────────────────────────────────────────┤
THEOREM Spec ⇒ □TypeOK   Checked by TLC.
PROOF OMITTED
├──────────────────────────────────────────────────────────────────────────┤
THEOREM Spec ⇒ ConsensusProperty   Checked by TLC.
PROOF OMITTED
├──────────────────────────────────────────────────────────────────────────┤
THEOREM Spec ⇒ CommonCoinProperty   Checked by TLC.
PROOF OMITTED
└──────────────────────────────────────────────────────────────────────────┘
```

Figure 33. The BenOrAbstByzCCLRstH TLA^{+} specification

MODULE *BenOrMsgsByzCC*

This specification is mostly the same as *BenOrMsgsByz* with some additional changes :
- Added a system action for tossing a common coin with a new variable *commonCoin*.
- Nodes can't estimate values randomly unless the common coin was tossed.
- All nodes randomly estimate the same value that is equal to the common coin value.
- A new temporal property *CommonCoinProperty* checks whether the coin is tossed for all rounds.

The common coin is used to model successful termination of the *BenOr Byzantine* consensus algorithm with all possible inputs given a number of nodes N.
With the common coin, *Byzantine* consensus is reached for all possible initial estimate values.

EXTENDS TLC, *Integers*, *FiniteSets*

Use the *BenOrByzCC.cfg* file.
CONSTANTS $NODES$, $FAULTY\_NODES$, $ESTIMATE\_ZERO$, $ESTIMATE\_ONE$, $CHECK\_ALL\_INITIAL\_VALUES$

VARIABLE *rounds*	Function that returns a given node's current round.
VARIABLE *estimates*	Function that returns a given node's current estimated value.
VARIABLE *decisions*	Function that returns a given node's current decision value.
VARIABLE *msgs*	Set of all sent messages.
VARIABLE *commonCoin*	Models the Common Coin.

$vars \triangleq \langle rounds, estimates, decisions, msgs, commonCoin \rangle$

For the *BenOr Byzantine* consensus protocol version, $N > 5 * F$ must be true. Also, F must be greater than 0, otherwise the $N - F$ check for proposal messages won't work.

ASSUME $\land (Cardinality(NODES) > 5 * Cardinality(FAULTY\_NODES))$
$\quad\quad\quad\land (Cardinality(FAULTY\_NODES) \geq 1)$

Nodes and their decisions must be specified.

ASSUME $\land NODES \neq \{\}$
$\quad\quad\quad\land ESTIMATE\_ONE \neq \{\}$
$\quad\quad\quad\land ESTIMATE\_ZERO \neq \{\}$

Decision values must be specified for all nodes.

ASSUME $Cardinality(ESTIMATE\_ZERO) + Cardinality(ESTIMATE\_ONE) = Cardinality(NODES)$

Numbers of nodes and faulty nodes.

$NumberOfNodes \triangleq Cardinality(NODES)$
$NumberOfFaultyNodes \triangleq Cardinality(FAULTY\_NODES)$

Set of node round numbers.

$NodeRounds \triangleq Nat$

Set of minimum rounds necessary to reach consensus with the Common Coin.

$CCRounds \triangleq 1 \,..\, 3$

Decision values of binary consensus.

$DecisionValues \triangleq \{\text{"0"}, \text{"1"}\}$

Set of all possible node decision values including the "-1" used for the initial state.

$NodeDecisionValues \triangleq DecisionValues \cup \{\text{"-1"}\}$

Set of all functions that map nodes to their estimate values.

$NodeEstimateFunctions \triangleq [NODES \rightarrow DecisionValues]$

Set of all possible node estimate values. The "-1" is not necessary since all nodes estimate a value of "0" or "1" in the initial state.

$NodeEstimateValues \triangleq DecisionValues$

Possible values and types of messages in the *BenOr* consensus algorithm.

$MessageType \triangleq \{\text{"report"}, \text{"proposal"}\}$
$MessageValues \triangleq \{\text{"0"}, \text{"1"}, \text{"?"}\}$

Each message record also has a round number.

$MessageRecordSet \triangleq [Sender : NODES, Round : Nat, Type : MessageType, Value : MessageValues]$

Set of all possible Common Coin values. "-1" means that the common coin was not tossed yet for that round.

$CommonCoinValues \triangleq \{\text{"-1"}, \text{"0"}, \text{"1"}\}$

Used to print variable values. Must be used with $\land$ and in states which are reached by TLC.

```
PrintVal(message, expression) ≜ Print(⟨message, expression⟩, TRUE)
```

A set of all nodes other than the specified node.

```
AllOtherNodes(node) ≜
    NODES \ {node}
```

Returns TRUE if a node sent the specified message during the specified round.

```
DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue) ≜
    LET messageRecord ≜ [Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ messageValue]
    IN  messageRecord ∈ msgs
```

Returns TRUE if a node sent a message of the specified type with any value ("0", "1", or "?") during the specified round.

```
DidNodeSendMessageOfType(sendingNode, nodeRound, messageType) ≜
    ∃ msgValue ∈ MessageValues :
        LET messageRecord ≜ [Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ msgValue]
        IN  messageRecord ∈ msgs
```

Adds a specific message to the set *msgs* with the sending node and the node's round. If the a message with the same value was already sent during the specified round, it is not added to
the *msgs* set.

```
SendBroadcastMessage(sendingNode, nodeRound, messageType, messageValue) ≜
    ∧ ¬DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue)
    ∧ msgs′ = msgs ∪ {[Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ messageValue]}
```

Returns the number of messages with the specified value that were sent by nodes other than *sendingNode*.

```
HowManyMessages(sendingNode, nodeRound, messageType, messageValue) ≜
    Cardinality({msg ∈ msgs :
                      ∧ msg.Sender ≠ sendingNode
                      ∧ msg.Round = nodeRound
                      ∧ msg.Type = messageType
                      ∧ msg.Value = messageValue})
```

Returns the number of messages sent by nodes other than *checkingNode* of the specified message type.

```
HowManyMessagesOfType(checkingNode, nodeRound, messageType) ≜
    Cardinality({node ∈ AllOtherNodes(checkingNode) :
        DidNodeSendMessageOfType(node, nodeRound, messageType)})
```

Returns a set of estimate values that are present in more than $(N + F)/\ 2$ report messages. Only one same value can be present in more than $(N + F)\ /\ 2$ messages.

```
MajorityOfReports(node, nodeRound) ≜
    {estimateValue ∈ NodeEstimateValues :
        HowManyMessages(node, nodeRound, "report", estimateValue) ∗ 2 > NumberOfNodes + NumberOfFaultyNodes}
```

Check if there are more than $N\ /\ 2$ report messages with the same value.

```
CheckPhase1(node, nodeRound) ≜
    MajorityOfReports(node, nodeRound) ≠ {}
```

After each node sends a report message during the first phase, it waits for $N - F$ report messages and then counts how many messages contain the same estimate value v.
If there are more than $(N + F)/\ 2$ of such messages, the node sends a proposal message with the value that was in more than $(N + F)/\ 2$ report messages.
Otherwise, the node sends a proposal message with the "?" value. All faulty nodes always send proposal messages with the "?" value.

```
WaitForReports(node) ≜
    ∧ node ∈ NODES
    ∧ DidNodeSendMessage(node, rounds[node], "report", estimates[node])
    ∧ HowManyMessagesOfType(node, rounds[node], "report") ≥ NumberOfNodes − NumberOfFaultyNodes
    ∧ IF  ∧ CheckPhase1(node, rounds[node])
          ∧ node ∉ FAULTY_NODES
       THEN ∃ majorityValue ∈ MajorityOfReports(node, rounds[node]) :
                 SendBroadcastMessage(node, rounds[node], "proposal", majorityValue)
       ELSE SendBroadcastMessage(node, rounds[node], "proposal", "?")
    ∧ UNCHANGED ⟨rounds, estimates, decisions, commonCoin⟩
```

The first phase of the *BenOr* consensus algorithm. All nodes send report messages with their estimate value v to all other nodes. Each node then checks what kind of values they have received from other nodes. The value v that is sent as a proposal message if it is in the majority of report messages
is not necessarily the same as the node's current estimate value. A node can send a value different from its own estimate value to other nodes in a proposal message.

```
$Phase1(node) \triangleq$
    $\land$ $node \in NODES$
    $\land$ $\lor$ $SendBroadcastMessage(node, rounds[node], \text{"report"}, estimates[node])$
      $\lor$ $WaitForReports(node)$
    $\land$ UNCHANGED $\langle rounds, estimates, decisions, commonCoin \rangle$
```

Returns a set of proposal values "0" or "1" that are proposed by more than $(N + F)$ / 2 nodes.

```
$MajorityOfProposals(node) \triangleq$
    $\{estimateValue \in NodeEstimateValues :$
        $HowManyMessages(node, rounds[node], \text{"proposal"}, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes\}$
```

Returns a set of proposal values "0" or "1" that are proposed by at least $F + 1$ nodes.

```
$AtLeastNF1Proposals(node) \triangleq$
    $\{estimateValue \in NodeEstimateValues :$
        $HowManyMessages(node, rounds[node], \text{"proposal"}, estimateValue) \geq NumberOfFaultyNodes + 1\}$
```

Check for more than $(N + F)$ / 2 proposal messages with the same value.

```
$CheckPhase2a(node) \triangleq$
    $MajorityOfProposals(node) \neq \{\}$
```

Check for $F + 1$ or more proposal messages with the same value.

```
$CheckPhase2b(node) \triangleq$
    $AtLeastNF1Proposals(node) \neq \{\}$
```

Decide an estimate value that has been proposed by more than $(N + F)$ / 2 nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$DecidePhase2a(node) \triangleq$
    $\exists\, decisionValue \in MajorityOfProposals(node) :$
        $decisions' = [decisions$ EXCEPT $![node] = decisionValue]$
```

Estimate a proposal value that has been proposed by at least $F + 1$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$EstimatePhase2b(node) \triangleq$
    $\exists\, estimateValue \in AtLeastNF1Proposals(node) :$
        $estimates' = [estimates$ EXCEPT $![node] = estimateValue]$
```

Pick a random estimate value from "0" or "1" for a node. *TLC* will check all possible choices separately.

```
$EstimatePhase2c(node) \triangleq$
    $\land$ $commonCoin[rounds[node]] \neq \text{"-1"}$
    $\land$ $estimates' = [estimates$ EXCEPT $![node] = commonCoin[rounds[node]]]$
```

A node must have sent a proposal message and must wait for $N - F$ proposal messages from other nodes.

```
$WaitForProposals(node) \triangleq$
    $\land$ $node \in NODES$
    $\land$ $DidNodeSendMessageOfType(node, rounds[node], \text{"proposal"})$
    $\land$ $HowManyMessagesOfType(node, rounds[node], \text{"proposal"}) \geq NumberOfNodes - NumberOfFaultyNodes$
```

The second phase of the *BenOr* consensus algorithm. All nodes that have sent a proposal message wait for $N - F$ of such messages from other nodes. If the node receives more than $(N + F)$ / 2 proposal messages with the same value v ("0" or "1"), then
the node sets its decision value to v. Furthermore, if the node receives at least least $F + 1$ proposal messages with the value v, it sets its estimate value to v. Otherwise, the node changes its estimate value to "0" or "1" randomly.

```
$Phase2(node) \triangleq$
    $\land$ $node \in NODES$
    $\land$ $WaitForProposals(node)$
    $\land$ IF $CheckPhase2a(node) \land CheckPhase2b(node)$
        THEN $DecidePhase2a(node) \land EstimatePhase2b(node)$
        ELSE IF $\neg CheckPhase2a(node) \land CheckPhase2b(node)$
            THEN $EstimatePhase2b(node) \land$ UNCHANGED $\langle decisions \rangle$
            ELSE IF $\neg CheckPhase2a(node) \land \neg CheckPhase2b(node)$
                THEN $EstimatePhase2c(node) \land$ UNCHANGED $\langle decisions \rangle$
```

```
                        ELSE  FALSE   No other cases possible.
   ∧ rounds′ = [rounds EXCEPT ![node] = @ + 1]     Node goes to the next round.
   ∧ UNCHANGED ⟨msgs, commonCoin⟩
```

Set the Common Coin value for rounds that have been reached by nodes. If in the round at least 1 node has already estimated some value non-randomly, then that round's common coin
value will be set to the node's estimate value (the Common Coin was lucky).
Otherwise, the Common Coin value is set to "0" or "1" and *TLC* will check behaviors when all nodes randomly estimate "0" or "1" together.

```
TossCommonCoin ≜
   ∧ ∃ round ∈ CCRounds :
      ∧ commonCoin[round] = "-1"
      ∧ ∨ ∃ node ∈ NODES \ FAULTY_NODES :   If at least 1 node already estimated some value in phase 2b.
               ∧ WaitForProposals(node)
               ∧ ∨ CheckPhase2a(node) ∧ CheckPhase2b(node)
                 ∨ ¬CheckPhase2a(node) ∧ CheckPhase2b(node)
               ∧ commonCoin′ = [commonCoin EXCEPT ![round] = estimates[node]]
        ∨ ∧ ∀ node ∈ NODES \ FAULTY_NODES :
               ∧ WaitForProposals(node)
               ∧ ¬CheckPhase2a(node)
               ∧ ¬CheckPhase2b(node)
          ∧ ∃ coinValue ∈ DecisionValues :   All nodes estimate both values based on the Common Coin.
               commonCoin′ = [commonCoin EXCEPT ![round] = coinValue]
   ∧ UNCHANGED ⟨rounds, estimates, decisions, msgs⟩
```

Consensus property for *TLC* to check. Consensus is guaranteed only between non-faulty nodes. All decision values must be "0" or "1" and all non-faulty nodes must decide
on the same value.

```
ConsensusReachedByzantine ≜
   ∧ ∀ node ∈ NODES \ FAULTY_NODES : decisions[node] ∈ DecisionValues
   ∧ ∀ node1, node2 ∈ NODES \ FAULTY_NODES : decisions[node1] = decisions[node2]
```

Common Coin validity property for *TLC* to check. The *CC* is valid, if its actually set during every round. Furthermore, it shouldn't take more than 3 rounds to reach consensus with a lucky *CC*.

```
CommonCoinValid ≜
   ∧ ∀ round ∈ CCRounds : commonCoin[round] ∈ DecisionValues
   ∧ ∀ node ∈ NODES : rounds[node] ≤ 3
```

The last state of the algorithm when consensus is reached.

```
LastState ≜
   ∧ ConsensusReachedByzantine

TypeOK ≜
   ∧   ∀ node ∈ NODES : rounds[node] ∈ NodeRounds
   ∧   ∀ node ∈ NODES : estimates[node] ∈ NodeEstimateValues
   ∧   ∀ node ∈ NODES : decisions[node] ∈ NodeDecisionValues
   ∧   msgs ⊆ MessageRecordSet
   ∧   ∀ round ∈ CCRounds : commonCoin[round] ∈ CommonCoinValues
```

If *CHECK_ALL_INITIAL_VALUES* is set to TRUE, *TLC* will check all possible estimate value combinations. Otherwise, initial estimates are defined by configuring what nodes estimate a "0" or a "1" value with the sets *ESTIMATE_ZERO* and *ESTIMATE_ONE*.
Initial decisions are set to "-1" at the start to ensure *ConsensusReached* is not true in the initial state.
There are no sent messages in the initial state.

```
Init ≜
   ∧ rounds = [node ∈ NODES ↦ 1]
   ∧ IF CHECK_ALL_INITIAL_VALUES
       THEN ∃ estimateFunction ∈ NodeEstimateFunctions : estimates = estimateFunction
       ELSE  estimates = [node ∈ NODES ↦ IF node ∈ ESTIMATE_ZERO THEN "0" ELSE "1"]
   ∧ decisions = [node ∈ NODES ↦ "-1"]
   ∧ msgs = {}
   ∧ commonCoin = [round ∈ CCRounds ↦ "-1"]

Next ≜
   ∧ ¬LastState  Stops TLC when used with deadlock check enabled.
   ∧ ∨ ∃ node ∈ NODES :
```

$\qquad\qquad \lor\ Phase1(node)$
$\qquad\qquad \lor\ Phase2(node)$
$\qquad \lor\ TossCommonCoin$

$Spec\ \triangleq$
$\quad \land\ Init$
$\quad \land\ \Box[Next]_{vars}$
$\quad \land\ \forall\, node \in NODES :$
$\qquad \mathrm{WF}_{vars}(Phase1(node) \lor Phase2(node))$
$\quad \land\ \mathrm{WF}_{vars}(TossCommonCoin)$

$ConsensusProperty\ \triangleq\ \Diamond\Box[ConsensusReachedByzantine]_{vars}$ Eventually, consensus will always be reached.
$CommonCoinProperty\ \triangleq\ \Diamond\Box[CommonCoinValid]_{vars}$ Eventually, the CC will be set for all rounds.

THEOREM $Spec \Rightarrow \Box\, TypeOK$ Checked by TLC.
PROOF OMITTED

THEOREM $Spec \Rightarrow ConsensusProperty$ Checked by TLC.
PROOF OMITTED

Figure 34. The BenOrMsgsByzCC TLA$^+$ specification

MODULE *BenOrMsgsByzCCH*

This is a refinement of *BenOrMsgsByzCC*, *BenOrAbstByzCC*, and *BenOrAbstByzCCH* with additional refinement history variables. Several modifications were required for *TLC* to successfully verify the refinement mapping *RefAbst* :

- Added the *estimateHistory* and *proposalHistory* variables that correspond to the *estimatesAtRound* and *proposalsAtRound* variables from *BenOrAbstByzCC*. All other variables are the same between both specifications and didn't require any additional changes.
- Modified the specification so that whenever a broadcast message is sent, the estimate or proposal value also changes the corresponding history variable.
- Made it so that whenever a node estimates a new value, then both estimates and *estimateHistory* are changed.
- Modified the *Init* formula to also set the *estimateHistory* values based on the initial estimates similarly to *BenOrAbstByzCC*.
- Defined a new action *SetNextRoundEstimate* that sets the *estimateHistory* values for the next round. This behavior is the same as the *SetNextRoundEstimate* action in *BenOrAbstByzCC*.
- Added an additional condition for the common coin action that requires for the *estimateHistory* variable to be set when tossing the coin if some node estimated some value. This is required for the refinement mapping *RefAbst*, but doesn't alter the state space in a way that would make the refinement mapping *RefMsgsCC* incorrect.

The refinement mappings *RefMsgsCC* and *RefAbstH* are also provided and can be checked with *TLC* if put inside the *RefinementProperty* definition before $'!Spec'$.

EXTENDS *TLC*, *Integers*, *FiniteSets*

Use the *BenOrByzCC.cfg* file.

CONSTANTS *NODES*, *FAULTY_NODES*, *ESTIMATE_ZERO*, *ESTIMATE_ONE*, *CHECK_ALL_INITIAL_VALUES*

VARIABLE *rounds* — Function that returns a given node's current round.
VARIABLE *estimates* — Function that returns a given node's current estimated value.
VARIABLE *decisions* — Function that returns a given node's current decision value.
VARIABLE *msgs* — Set of all sent messages.
VARIABLE *commonCoin* — Models the Common Coin.
VARIABLE *estimateHistory* — History variables for refinement.
VARIABLE *proposalHistory*

$vars \triangleq \langle rounds, estimates, decisions, msgs, commonCoin, estimateHistory, proposalHistory \rangle$

For the *BenOr Byzantine* consensus protocol version, $N > 5 * F$ must be true. Also, F must be greater than 0, otherwise the $N - F$ check for proposal messages won't work.

ASSUME $\land (Cardinality(NODES) > 5 * Cardinality(FAULTY\_NODES))$
$\qquad\land (Cardinality(FAULTY\_NODES) \geq 1)$

Nodes and their decisions must be specified.

ASSUME $\land NODES \neq \{\}$
$\qquad\land ESTIMATE\_ONE \neq \{\}$
$\qquad\land ESTIMATE\_ZERO \neq \{\}$

Decision values must be specified for all nodes.

ASSUME $Cardinality(ESTIMATE\_ZERO) + Cardinality(ESTIMATE\_ONE) = Cardinality(NODES)$

Numbers of nodes and faulty nodes.

$NumberOfNodes \triangleq Cardinality(NODES)$
$NumberOfFaultyNodes \triangleq Cardinality(FAULTY\_NODES)$

Set of node round numbers.

$NodeRounds \triangleq Nat$

Set of minimum rounds necessary to reach consensus with the Common Coin.

$CCRounds \triangleq 1 \,..\, 3$

Decision values of binary consensus.

$DecisionValues \triangleq \{\text{"0"}, \text{"1"}\}$

Set of all possible node decision values including the "-1" used for the initial state.

$NodeDecisionValues \triangleq DecisionValues \cup \{\text{"-1"}\}$

Set of all functions that map nodes to their estimate values.

$NodeEstimateFunctions \triangleq [NODES \rightarrow DecisionValues]$

Set of all possible node estimate values. The "-1" is not necessary since all nodes estimate a value of "0" or "1" in the initial state.

$NodeEstimateValues \triangleq DecisionValues$

Set of all functions that map nodes to their estimate values and "-1".

$AllNodeEstimateFunctions \triangleq [NODES \rightarrow NodeDecisionValues]$

Possible values and types of messages in the *BenOr* consensus algorithm.

```
MessageType ≜ {"report", "proposal"}
MessageValues ≜ {"0", "1", "?"}
```

Set of all functions that map nodes to their proposal message values and "-1".

```
AllNodeProposalFunctions ≜ [NODES → MessageValues ∪ {"-1"}]
```

Each message record also has a round number.

```
MessageRecordSet ≜ [Sender : NODES, Round : Nat, Type : MessageType, Value : MessageValues]
```

Set of all possible Common Coin values. "-1" means that the common coin was not tossed yet for that round.

```
CommonCoinValues ≜ {"-1", "0", "1"}
```

Used to print variable values. Must be used with ∧ and in states which are reached by *TLC*.

```
PrintVal(message, expression) ≜ Print(⟨message, expression⟩, TRUE)
```

A set of all nodes other than the specified node.

```
AllOtherNodes(node) ≜
   NODES \ {node}
```

Returns TRUE if a node sent the specified message during the specified round.

```
DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue) ≜
   LET messageRecord ≜ [Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ messageValue]
   IN  messageRecord ∈ msgs
```

Returns TRUE if a node sent a message of the specified type with any value ("0", "1", or "?") during the specified round.

```
DidNodeSendMessageOfType(sendingNode, nodeRound, messageType) ≜
   ∃ msgValue ∈ MessageValues :
      LET messageRecord ≜ [Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ msgValue]
      IN  messageRecord ∈ msgs
```

Adds a specific message to the set *msgs* with the sending node and the node's round. If the a message with the same value was already sent during the specified round, it is not added to
the *msgs* set.

```
SendBroadcastMessage(sendingNode, nodeRound, messageType, messageValue) ≜
   ∧ ¬DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue)
   ∧ msgs' = msgs ∪ {[Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ messageValue]}
   ∧ IF messageType = "report"
      THEN estimateHistory' = [estimateHistory EXCEPT ![nodeRound][sendingNode] = messageValue]
      ELSE UNCHANGED ⟨estimateHistory⟩
   ∧ IF messageType = "proposal"
      THEN proposalHistory' = [proposalHistory EXCEPT ![nodeRound][sendingNode] = messageValue]
      ELSE UNCHANGED ⟨proposalHistory⟩
```

Returns the number of messages with the specified value that were sent by nodes other than *sendingNode*.

```
HowManyMessages(sendingNode, nodeRound, messageType, messageValue) ≜
   Cardinality({msg ∈ msgs :
                  ∧ msg.Sender ≠ sendingNode
                  ∧ msg.Round = nodeRound
                  ∧ msg.Type = messageType
                  ∧ msg.Value = messageValue})
```

Returns the number of messages sent by nodes other than *checkingNode* of the specified message type.

```
HowManyMessagesOfType(checkingNode, nodeRound, messageType) ≜
   Cardinality({node ∈ AllOtherNodes(checkingNode) :
      DidNodeSendMessageOfType(node, nodeRound, messageType)})
```

Returns a set of estimate values that are present in more than $(N + F)/ 2$ report messages. Only one same value can be present in more than $(N + F) / 2$ messages.

```
MajorityOfReports(node, nodeRound) ≜
   {estimateValue ∈ NodeEstimateValues :
      HowManyMessages(node, nodeRound, "report", estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes}
```

Check if there are more than $N / 2$ report messages with the same value.

```
CheckPhase1(node, nodeRound) ≜
```

```
    $MajorityOfReports(node, nodeRound) \neq \{\}$
```

After each node sends a report message during the first phase, it waits for $N - F$ report messages and then counts how many messages contain the same estimate value v.
If there are more than $(N + F)/\ 2$ of such messages, the node sends a proposal message with the value that was in more than $(N + F)/\ 2$ report messages.
Otherwise, the node sends a proposal message with the "?" value. All faulty nodes always send proposal messages with the "?" value.

```
$WaitForReports(node) \triangleq$
    $\land\ node \in NODES$
    $\land\ DidNodeSendMessage(node, rounds[node], \text{"report"}, estimates[node])$
    $\land\ HowManyMessagesOfType(node, rounds[node], \text{"report"}) \geq NumberOfNodes - NumberOfFaultyNodes$
    $\land$ IF $\land\ CheckPhase1(node, rounds[node])$
            $\land\ node \notin FAULTY\_NODES$
       THEN $\exists\ majorityValue \in MajorityOfReports(node, rounds[node]) :$
                 $SendBroadcastMessage(node, rounds[node], \text{"proposal"}, majorityValue)$
       ELSE $SendBroadcastMessage(node, rounds[node], \text{"proposal"}, \text{"?"})$
    $\land$ UNCHANGED $\langle rounds, estimates, decisions, commonCoin \rangle$
```

The first phase of the *BenOr* consensus algorithm. All nodes send report messages with their estimate value v to all other nodes. Each node then checks what kind of values they have received from other nodes. The value v that is sent as a proposal message if it is in the majority of report messages
is not necessarily the same as the node's current estimate value. A node can send a value different from its own estimate value to other nodes in a proposal message.

```
$Phase1(node) \triangleq$
    $\land\ node \in NODES$
    $\land\ \lor\ SendBroadcastMessage(node, rounds[node], \text{"report"}, estimates[node])$
       $\lor\ WaitForReports(node)$
    $\land$ UNCHANGED $\langle rounds, estimates, decisions, commonCoin \rangle$
```

Returns a set of proposal values "0" or "1" that are proposed by more than $(N + F)\ /\ 2$ nodes.

```
$MajorityOfProposals(node) \triangleq$
    $\{estimateValue \in NodeEstimateValues :$
        $HowManyMessages(node, rounds[node], \text{"proposal"}, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes\}$
```

Returns a set of proposal values "0" or "1" that are proposed by at least $F + 1$ nodes.

```
$AtLeastNF1Proposals(node) \triangleq$
    $\{estimateValue \in NodeEstimateValues :$
        $HowManyMessages(node, rounds[node], \text{"proposal"}, estimateValue) \geq NumberOfFaultyNodes + 1\}$
```

Check for more than $(N + F)\ /\ 2$ proposal messages with the same value.

```
$CheckPhase2a(node) \triangleq$
    $MajorityOfProposals(node) \neq \{\}$
```

Check for $F + 1$ or more proposal messages with the same value.

```
$CheckPhase2b(node) \triangleq$
    $AtLeastNF1Proposals(node) \neq \{\}$
```

Decide an estimate value that has been proposed by more than $(N + F)\ /\ 2$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$DecidePhase2a(node) \triangleq$
    $\exists\ decisionValue \in MajorityOfProposals(node) :$
        $decisions' = [decisions$ EXCEPT $![node] = decisionValue]$
```

Estimate a proposal value that has been proposed by at least $F + 1$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$EstimatePhase2b(node) \triangleq$
    $\exists\ estimateValue \in AtLeastNF1Proposals(node) :$
        $\land\ estimates' = [estimates$ EXCEPT $![node] = estimateValue]$
        $\land\ estimateHistory' = [estimateHistory$ EXCEPT $![rounds[node]][node] = estimateValue]$
```

Pick a random estimate value from "0" or "1" for a node. *TLC* will check all possible choices separately.

```
$EstimatePhase2c(node) \triangleq$
    $\land\ commonCoin[rounds[node]] \neq \text{"-1"}$
    $\land\ estimates' = [estimates$ EXCEPT $![node] = commonCoin[rounds[node]]]$
    $\land\ estimateHistory' = [estimateHistory$ EXCEPT $![rounds[node]][node] = commonCoin[rounds[node]]]$
```

```
A node must have sent a proposal message and must wait for N − F proposal messages from other nodes.
WaitForProposals(node) ≜
    ∧ node ∈ NODES
    ∧ DidNodeSendMessageOfType(node, rounds[node], "proposal")
    ∧ HowManyMessagesOfType(node, rounds[node], "proposal") ≥ NumberOfNodes − NumberOfFaultyNodes

The second phase of the BenOr consensus algorithm. All nodes that have sent a proposal message wait for N − F of such messages from
other nodes. If the node receives more than (N + F) / 2 proposal messages with the same value v ("0" or "1"), then
 the node sets its decision value to v. Furthermore, if the node receives at least least F + 1 proposal messages with the value v, it sets its
 estimate value to v. Otherwise, the node changes its estimate value to "0" or "1" randomly.
Phase2(node) ≜
    ∧ node ∈ NODES
    ∧ WaitForProposals(node)
    ∧ IF CheckPhase2a(node) ∧ CheckPhase2b(node)
        THEN DecidePhase2a(node) ∧ EstimatePhase2b(node)
        ELSE IF ¬CheckPhase2a(node) ∧ CheckPhase2b(node)
               THEN EstimatePhase2b(node) ∧ UNCHANGED ⟨decisions⟩
               ELSE IF ¬CheckPhase2a(node) ∧ ¬CheckPhase2b(node)
                      THEN EstimatePhase2c(node) ∧ UNCHANGED ⟨decisions⟩
                      ELSE FALSE   No other cases possible.
    ∧ rounds' = [rounds EXCEPT ![node] = @ + 1]   Node goes to the next round.
    ∧ UNCHANGED ⟨msgs, commonCoin, proposalHistory⟩

Set the Common Coin value for the current round. If in the round at least 1 node has already estimated some value non-randomly, then
that round's common coin
 value will be set to the node's estimate value (the Common Coin was lucky).
Otherwise, the Common Coin value is set to "0" or "1" and TLC will check behaviors when all nodes randomly estimate "0" or "1" together.
TossCommonCoin ≜
    ∧ ∃ round ∈ CCRounds :
        ∧ commonCoin[round] = "-1"
        ∧ ∨ ∃ node ∈ NODES \ FAULTY_NODES :   If at least 1 node already estimated some value in phase 2b.
              ∧ estimateHistory[round][node] ≠ "-1"
              ∧ WaitForProposals(node)
              ∧ ∨ CheckPhase2a(node) ∧ CheckPhase2b(node)
                ∨ ¬CheckPhase2a(node) ∧ CheckPhase2b(node)
              ∧ commonCoin' = [commonCoin EXCEPT ![round] = estimateHistory[round][node]]
          ∨ ∧ ∀ node ∈ NODES \ FAULTY_NODES :
                ∧ WaitForProposals(node)
                ∧ ¬CheckPhase2a(node)
                ∧ ¬CheckPhase2b(node)
            ∧ ∃ coinValue ∈ DecisionValues :   All nodes estimate both values based on the Common Coin.
                commonCoin' = [commonCoin EXCEPT ![round] = coinValue]
    ∧ UNCHANGED ⟨rounds, estimates, decisions, msgs, estimateHistory, proposalHistory⟩

Set the node's next round's estimate to the previous round's estimate value so that other nodes are aware that the node has entered the
 next round. In the previous round the estimate will be some valid value other than "-1".
SetNextRoundEstimate(node) ≜
    ∧ node ∈ NODES
    ∧ estimateHistory[rounds[node]][node] = "-1"
    ∧ estimateHistory' = [estimateHistory EXCEPT ![rounds[node]][node] = estimateHistory[rounds[node] − 1][node]]
    ∧ SendBroadcastMessage(node, rounds[node], "report", estimates[node])
    ∧ UNCHANGED ⟨rounds, estimates, decisions, commonCoin, proposalHistory⟩

Consensus property for TLC to check. Consensus is guaranteed only between non-faulty nodes. All decision values must be "0" or "1" and
all non-faulty nodes must decide
 on the same value.
ConsensusReachedByzantine ≜
    ∧ ∀ node ∈ NODES \ FAULTY_NODES : decisions[node] ∈ DecisionValues
    ∧ ∀ node1, node2 ∈ NODES \ FAULTY_NODES : decisions[node1] = decisions[node2]

Common Coin validity property for TLC to check. The CC is valid, if its actually set during every round. Furthermore, it shouldn't take
more than 3 rounds to reach consensus with a lucky CC.
```

```
$CommonCoinValid \triangleq$
    $\land \forall round \in CCRounds : commonCoin[round] \in DecisionValues$
    $\land \forall node \in NODES : rounds[node] \leq 3$
```

The last state of the algorithm when consensus is reached.

```
$LastState \triangleq$
    $\land ConsensusReachedByzantine$

$TypeOK \triangleq$
    $\land \quad \forall round \in CCRounds : estimateHistory[round] \in AllNodeEstimateFunctions$
    $\land \quad \forall round \in CCRounds : proposalHistory[round] \in AllNodeProposalFunctions$
    $\land \quad \forall node \in NODES : rounds[node] \in NodeRounds$
    $\land \quad \forall node \in NODES : estimates[node] \in NodeEstimateValues$
    $\land \quad \forall node \in NODES : decisions[node] \in NodeDecisionValues$
    $\land \quad msgs \subseteq MessageRecordSet$
    $\land \quad \forall round \in CCRounds : commonCoin[round] \in CommonCoinValues$
```

If *CHECK_ALL_INITIAL_VALUES* is set to TRUE, *TLC* will check all possible estimate value combinations. Otherwise, initial estimates are defined by configuring what nodes estimate a "0" or a "1" value with the sets *ESTIMATE_ZERO* and *ESTIMATE_ONE*.
Initial decisions are set to "-1" at the start to ensure *ConsensusReached* is not true in the initial state.
There are no sent messages in the initial state.

```
$Init \triangleq$
    $\land$ IF $CHECK\_ALL\_INITIAL\_VALUES$
       THEN $\exists estimateFunction \in NodeEstimateFunctions :$
              $\land estimateHistory = [round \in CCRounds \mapsto$ IF $round = 1$
                                                            THEN $estimateFunction$
                                                            ELSE $[node \in NODES \mapsto$ "-1"$]]$
              $\land estimates = estimateFunction$
       ELSE   $\land estimateHistory = [round \in CCRounds \mapsto$ IF $round = 1$
                                                            THEN $[node \in NODES \mapsto$
                                                               IF $node \in ESTIMATE\_ZERO$
                                                                THEN "0"
                                                                ELSE "1"$]$
                                                            ELSE $[node \in NODES \mapsto$ "-1"$]]$
              $\land estimates = [node \in NODES \mapsto$ IF $node \in ESTIMATE\_ZERO$ THEN "0" ELSE "1"$]$
    $\land proposalHistory = [round \in CCRounds \mapsto [node \in NODES \mapsto$ "-1"$]]$
    $\land rounds = [node \in NODES \mapsto 1]$
    $\land decisions = [node \in NODES \mapsto$ "-1"$]$
    $\land msgs = \{\}$
    $\land commonCoin = [round \in CCRounds \mapsto$ "-1"$]$

$Next \triangleq$
    $\land \neg LastState$   Stops $TLC$ when used with deadlock check enabled.
    $\land \lor \exists node \in NODES :$
            $\lor Phase1(node)$
            $\lor Phase2(node)$
            $\lor SetNextRoundEstimate(node)$
       $\lor TossCommonCoin$

$Spec \triangleq$
    $\land Init$
    $\land \Box[Next]_{vars}$
    $\land \forall node \in NODES :$
        $\mathrm{WF}_{vars}(Phase1(node) \lor Phase2(node) \lor SetNextRoundEstimate(node))$
    $\land \mathrm{WF}_{vars}(TossCommonCoin)$
```

VERY IMPORTANT: when checking the *RefinementProperty*, all INSTANCE statements in the other refined history variable specification *.tla* file must be commented out to avoid circular dependencies.
Also, the *RefinementProperty* definition must be commented out as well since the INSTANCE statements would not be defined in such a case.

```
$RefAbstH \triangleq$
   INSTANCE $BenOrAbstByzCCH$
    WITH $msgsHistory \leftarrow msgs,$
         $estimateHistory \leftarrow estimates,$
         $estimatesAtRound \leftarrow estimateHistory,$
```

```
            proposalsAtRound ← proposalHistory

RefMsgsCC ≜
    INSTANCE BenOrMsgsByzCC

RefAbst ≜
    INSTANCE BenOrAbstByzCC
     WITH estimatesAtRound ← estimateHistory,
            proposalsAtRound ← proposalHistory

RefinementProperty ≜    Can be uncommented here and in the BenOrByzCC.cfg file to check refinement.
     ∧ RefAbst!Spec
     ∧ RefAbstH!Spec
     ∧ RefMsgsCC!Spec
ConsensusProperty ≜ ◇□[ConsensusReachedByzantine]_vars  Eventually, consensus will always be reached.
CommonCoinProperty ≜ ◇□[CommonCoinValid]_vars  Eventually, the CC will be set for all rounds.
├──────────────────────────────────────────────────────────────────────────────┤
THEOREM Spec ⇒ □TypeOK  Checked by TLC.
PROOF OMITTED
├──────────────────────────────────────────────────────────────────────────────┤
```

Figure 35. The BenOrMsgsByzCCH TLA$^+$ specification

MODULE *BenOrMsgsByzCCL*

This specification is a refinement of *BenOrMsgsByzCC* that contains a node action for tossing the common coin. Now instead of the coin being tossed by the system, some node tosses the coin.
The refinement mapping *RefMsgsCC* is also included in the specification and can be checked with *TLC*. The specification contains fewer distinct states compared to *BenOrMsgsByzCC*, but as the number of nodes increase the amount of total states becomes greater than *BenOrMsgsByzCC*.

EXTENDS TLC, $Integers$, $FiniteSets$

Use the *BenOrByzCC.cfg* file.

CONSTANTS $NODES$, $FAULTY\_NODES$, $ESTIMATE\_ZERO$, $ESTIMATE\_ONE$, $CHECK\_ALL\_INITIAL\_VALUES$

VARIABLE $rounds$	Function that returns a given node's current round.
VARIABLE $estimates$	Function that returns a given node's current estimated value.
VARIABLE $decisions$	Function that returns a given node's current decision value.
VARIABLE $msgs$	Set of all sent messages.
VARIABLE $commonCoin$	Models the Common Coin.

$vars \triangleq \langle rounds, estimates, decisions, msgs, commonCoin \rangle$

For the *BenOr Byzantine* consensus protocol version, $N > 5 * F$ must be true. Also, F must be greater than 0, otherwise the $N - F$ check for proposal messages won't work.

ASSUME $\land (Cardinality(NODES) > 5 * Cardinality(FAULTY\_NODES))$
$\quad\quad\quad\land (Cardinality(FAULTY\_NODES) \geq 1)$

Nodes and their decisions must be specified.

ASSUME $\land NODES \neq \{\}$
$\quad\quad\quad\land ESTIMATE\_ONE \neq \{\}$
$\quad\quad\quad\land ESTIMATE\_ZERO \neq \{\}$

Decision values must be specified for all nodes.

ASSUME $Cardinality(ESTIMATE\_ZERO) + Cardinality(ESTIMATE\_ONE) = Cardinality(NODES)$

Numbers of nodes and faulty nodes.

$NumberOfNodes \triangleq Cardinality(NODES)$
$NumberOfFaultyNodes \triangleq Cardinality(FAULTY\_NODES)$

Set of node round numbers.

$NodeRounds \triangleq Nat$

Set of minimum rounds necessary to reach consensus with the Common Coin.

$CCRounds \triangleq 1 \,..\, 3$

Decision values of binary consensus.

$DecisionValues \triangleq \{\text{"0"}, \text{"1"}\}$

Set of all possible node decision values including the "-1" used for the initial state.

$NodeDecisionValues \triangleq DecisionValues \cup \{\text{"-1"}\}$

Set of all functions that map nodes to their estimate values.

$NodeEstimateFunctions \triangleq [NODES \rightarrow DecisionValues]$

Set of all possible node estimate values. The "-1" is not necessary since all nodes estimate a value of "0" or "1" in the initial state.

$NodeEstimateValues \triangleq DecisionValues$

Possible values and types of messages in the *BenOr* consensus algorithm.

$MessageType \triangleq \{\text{"report"}, \text{"proposal"}\}$
$MessageValues \triangleq \{\text{"0"}, \text{"1"}, \text{"?"}\}$

Each message record also has a round number.

$MessageRecordSet \triangleq [Sender : NODES, Round : Nat, Type : MessageType, Value : MessageValues]$

Set of all possible Common Coin values. "-1" means that the common coin was not tossed yet for that round.

$CommonCoinValues \triangleq \{\text{"-1"}, \text{"0"}, \text{"1"}\}$

Used to print variable values. Must be used with $\land$ and in states which are reached by *TLC*.

$PrintVal(message, expression) \triangleq Print(\langle message, expression \rangle, \text{TRUE})$

A set of all nodes other than the specified node.

```
$AllOtherNodes(node) \triangleq$
    $NODES \setminus \{node\}$
```

Returns TRUE if a node sent the specified message during the specified round.

```
$DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue) \triangleq$
    LET $messageRecord \triangleq [Sender \mapsto sendingNode, Round \mapsto nodeRound, Type \mapsto messageType, Value \mapsto messageValue]$
    IN  $messageRecord \in msgs$
```

Returns TRUE if a node sent a message of the specified type with any value ("0", "1", or "?") during the specified round.

```
$DidNodeSendMessageOfType(sendingNode, nodeRound, messageType) \triangleq$
    $\exists\, msgValue \in MessageValues :$
        LET $messageRecord \triangleq [Sender \mapsto sendingNode, Round \mapsto nodeRound, Type \mapsto messageType, Value \mapsto msgValue]$
        IN  $messageRecord \in msgs$
```

Adds a specific message to the set *msgs* with the sending node and the node's round. If the a message with the same value was already sent during the specified round, it is not added to
the *msgs* set.

```
$SendBroadcastMessage(sendingNode, nodeRound, messageType, messageValue) \triangleq$
    $\land \neg DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue)$
    $\land msgs' = msgs \cup \{[Sender \mapsto sendingNode, Round \mapsto nodeRound, Type \mapsto messageType, Value \mapsto messageValue]\}$
```

Returns the number of messages with the specified value that were sent by nodes other than *sendingNode*.

```
$HowManyMessages(sendingNode, nodeRound, messageType, messageValue) \triangleq$
    $Cardinality(\{msg \in msgs :$
                    $\land msg.Sender \neq sendingNode$
                    $\land msg.Round = nodeRound$
                    $\land msg.Type = messageType$
                    $\land msg.Value = messageValue\})$
```

Returns the number of messages sent by nodes other than *checkingNode* of the specified message type.

```
$HowManyMessagesOfType(checkingNode, nodeRound, messageType) \triangleq$
    $Cardinality(\{node \in AllOtherNodes(checkingNode) :$
        $DidNodeSendMessageOfType(node, nodeRound, messageType)\})$
```

Returns a set of estimate values that are present in more than $(N + F)/ 2$ report messages. Only one same value can be present in more than $(N + F)\ /\ 2$ messages.

```
$MajorityOfReports(node, nodeRound) \triangleq$
    $\{estimateValue \in NodeEstimateValues :$
        $HowManyMessages(node, nodeRound, \text{"report"}, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes\}$
```

Check if there are more than $N\ /\ 2$ report messages with the same value.

```
$CheckPhase1(node, nodeRound) \triangleq$
    $MajorityOfReports(node, nodeRound) \neq \{\}$
```

After each node sends a report message during the first phase, it waits for $N - F$ report messages and then counts how many messages contain the same estimate value v.
If there are more than $(N + F)/ 2$ of such messages, the node sends a proposal message with the value that was in more than $(N + F)/ 2$ report messages.
Otherwise, the node sends a proposal message with the "?" value. All faulty nodes always send proposal messages with the "?" value.

```
$WaitForReports(node) \triangleq$
    $\land node \in NODES$
    $\land DidNodeSendMessage(node, rounds[node], \text{"report"}, estimates[node])$
    $\land HowManyMessagesOfType(node, rounds[node], \text{"report"}) \geq NumberOfNodes - NumberOfFaultyNodes$
    $\land$ IF $\land CheckPhase1(node, rounds[node])$
           $\land node \notin FAULTY\_NODES$
       THEN $\exists\, majorityValue \in MajorityOfReports(node, rounds[node]) :$
                $SendBroadcastMessage(node, rounds[node], \text{"proposal"}, majorityValue)$
       ELSE $SendBroadcastMessage(node, rounds[node], \text{"proposal"}, \text{"?"})$
    $\land$ UNCHANGED $\langle rounds, estimates, decisions, commonCoin \rangle$
```

The first phase of the *BenOr* consensus algorithm. All nodes send report messages with their estimate value v to all other nodes. Each node then checks what kind of values they have received from other nodes. The value v that is sent as a proposal message if it is in the majority of report messages
is not necessarily the same as the node's current estimate value. A node can send a value different from its own estimate value to other nodes in a proposal message.

```
$Phase1(node) \triangleq$
    $\land\ node \in NODES$
    $\land\ \lor\ SendBroadcastMessage(node, rounds[node],$ “report”, $estimates[node])$
       $\lor\ WaitForReports(node)$
    $\land$ UNCHANGED $\langle rounds, estimates, decisions, commonCoin \rangle$
```

Returns a set of proposal values “0” or “1” that are proposed by more than $(N + F)$ / 2 nodes.

```
$MajorityOfProposals(node) \triangleq$
    $\{estimateValue \in NodeEstimateValues :$
        $HowManyMessages(node, rounds[node],$ “proposal”, $estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes\}$
```

Returns a set of proposal values “0” or “1” that are proposed by at least $F + 1$ nodes.

```
$AtLeastNF1Proposals(node) \triangleq$
    $\{estimateValue \in NodeEstimateValues :$
        $HowManyMessages(node, rounds[node],$ “proposal”, $estimateValue) \geq NumberOfFaultyNodes + 1\}$
```

Check for more than $(N + F)$ / 2 proposal messages with the same value.

```
$CheckPhase2a(node) \triangleq$
    $MajorityOfProposals(node) \neq \{\}$
```

Check for $F + 1$ or more proposal messages with the same value.

```
$CheckPhase2b(node) \triangleq$
    $AtLeastNF1Proposals(node) \neq \{\}$
```

Decide an estimate value that has been proposed by more than $(N + F)$ / 2 nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$DecidePhase2a(node) \triangleq$
    $\exists\ decisionValue \in MajorityOfProposals(node) :$
        $decisions' = [decisions$ EXCEPT $![node] = decisionValue]$
```

Estimate a proposal value that has been proposed by at least $F + 1$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$EstimatePhase2b(node) \triangleq$
    $\exists\ estimateValue \in AtLeastNF1Proposals(node) :$
        $estimates' = [estimates$ EXCEPT $![node] = estimateValue]$
```

Pick a random estimate value from “0” or “1” for a node. *TLC* will check all possible choices separately.

```
$EstimatePhase2c(node) \triangleq$
    $\land\ commonCoin[rounds[node]] \neq$ “-1”
    $\land\ estimates' = [estimates$ EXCEPT $![node] = commonCoin[rounds[node]]]$
```

A node must have sent a proposal message and must wait for $N - F$ proposal messages from other nodes.

```
$WaitForProposals(node) \triangleq$
    $\land\ node \in NODES$
    $\land\ DidNodeSendMessageOfType(node, rounds[node],$ “proposal”$)$
    $\land\ HowManyMessagesOfType(node, rounds[node],$ “proposal”$) \geq NumberOfNodes - NumberOfFaultyNodes$
```

The second phase of the *BenOr* consensus algorithm. All nodes that have sent a proposal message wait for $N - F$ of such messages from other nodes. If the node receives more than $(N + F)$ / 2 proposal messages with the same value v (“0” or “1”), then
the node sets its decision value to v. Furthermore, if the node receives at least least $F + 1$ proposal messages with the value v, it sets its estimate value to v. Otherwise, the node changes its estimate value to “0” or “1” randomly.

```
$Phase2(node) \triangleq$
    $\land\ node \in NODES$
    $\land\ WaitForProposals(node)$
    $\land$ IF $CheckPhase2a(node) \land CheckPhase2b(node)$
        THEN $DecidePhase2a(node) \land EstimatePhase2b(node)$
        ELSE IF $\neg CheckPhase2a(node) \land CheckPhase2b(node)$
            THEN $EstimatePhase2b(node) \land$ UNCHANGED $\langle decisions \rangle$
            ELSE IF $\neg CheckPhase2a(node) \land \neg CheckPhase2b(node)$
                THEN $EstimatePhase2c(node) \land$ UNCHANGED $\langle decisions \rangle$
                ELSE FALSE   No other cases possible.
    $\land\ rounds' = [rounds$ EXCEPT $![node] = @ + 1]$   Node goes to the next round.
    $\land$ UNCHANGED $\langle msgs, commonCoin \rangle$
```

A node sets the Common Coin value for the current round the node is in. If in the round at least 1 node has already estimated some value non-randomly, then that round's common coin
value will be set to the node's estimate value (the Common Coin was lucky).
Otherwise, the Common Coin value is set to "0" or "1" and *TLC* will check behaviors when all nodes randomly estimate "0" or "1" together.

```
$TossCommonCoin(settingNode) \triangleq$
    $\land$ $commonCoin[rounds[settingNode]] =$ "-1"
    $\land$ $\lor \exists node \in NODES \setminus FAULTY\_NODES :$   If at least 1 node already estimated some value in phase 2b.
                $\land$ $WaitForProposals(node)$
                $\land$ $\lor CheckPhase2a(node) \land CheckPhase2b(node)$
                   $\lor \neg CheckPhase2a(node) \land CheckPhase2b(node)$
                $\land$ $commonCoin' = [commonCoin$ EXCEPT $![rounds[settingNode]] = estimates[node]]$
       $\lor$ $\land \forall node \in NODES \setminus FAULTY\_NODES :$
                $\land$ $WaitForProposals(node)$
                $\land$ $\neg CheckPhase2a(node)$
                $\land$ $\neg CheckPhase2b(node)$
          $\land$ $\exists coinValue \in DecisionValues :$   All nodes estimate both values based on the Common Coin.
                $commonCoin' = [commonCoin$ EXCEPT $![rounds[settingNode]] = coinValue]$
    $\land$ UNCHANGED $\langle rounds, estimates, decisions, msgs \rangle$
```

Consensus property for *TLC* to check. Consensus is guaranteed only between non-faulty nodes. All decision values must be "0" or "1" and all non-faulty nodes must decide
on the same value.

```
$ConsensusReachedByzantine \triangleq$
    $\land \forall node \in NODES \setminus FAULTY\_NODES : decisions[node] \in DecisionValues$
    $\land \forall node1, node2 \in NODES \setminus FAULTY\_NODES : decisions[node1] = decisions[node2]$
```

Common Coin validity property for *TLC* to check. The *CC* is valid, if its actually set during every round. Furthermore, it shouldn't take more than 3 rounds to reach consensus with a lucky *CC*.

```
$CommonCoinValid \triangleq$
    $\land \forall round \in CCRounds : commonCoin[round] \in DecisionValues$
    $\land \forall node \in NODES : rounds[node] \leq 3$
```

The last state of the algorithm when consensus is reached.

```
$LastState \triangleq$
    $\land ConsensusReachedByzantine$

$TypeOK \triangleq$
    $\land$   $\forall node \in NODES : rounds[node] \in NodeRounds$
    $\land$   $\forall node \in NODES : estimates[node] \in NodeEstimateValues$
    $\land$   $\forall node \in NODES : decisions[node] \in NodeDecisionValues$
    $\land$   $msgs \subseteq MessageRecordSet$
    $\land$   $\forall round \in CCRounds : commonCoin[round] \in CommonCoinValues$
```

If *CHECK_ALL_INITIAL_VALUES* is set to TRUE, *TLC* will check all possible estimate value combinations. Otherwise, initial estimates are defined by configuring what nodes estimate a "0" or a "1" value with the sets *ESTIMATE_ZERO* and *ESTIMATE_ONE*.
Initial decisions are set to "-1" at the start to ensure *ConsensusReached* is not true in the initial state.
There are no sent messages in the initial state.

```
$Init \triangleq$
    $\land rounds = [node \in NODES \mapsto 1]$
    $\land$ IF $CHECK\_ALL\_INITIAL\_VALUES$
         THEN $\exists estimateFunction \in NodeEstimateFunctions : estimates = estimateFunction$
         ELSE  $estimates = [node \in NODES \mapsto$ IF $node \in ESTIMATE\_ZERO$ THEN "0" ELSE "1"$]$
    $\land decisions = [node \in NODES \mapsto$ "-1"$]$
    $\land msgs = \{\}$
    $\land commonCoin = [round \in CCRounds \mapsto$ "-1"$]$

$Next \triangleq$
    $\exists node \in NODES :$
       $\land \neg LastState$  Stops TLC when used with deadlock check enabled.
       $\land \lor Phase1(node)$
          $\lor Phase2(node)$
          $\lor TossCommonCoin(node)$

$Spec \triangleq$
```

```
    $\land$ $Init$
    $\land$ $\Box[Next]_{vars}$
    $\land$ $\forall\, node \in NODES$ :
        $\mathrm{WF}_{vars}(Phase1(node) \lor Phase2(node) \lor TossCommonCoin(node))$

$RefMsgsCC \triangleq$
    INSTANCE $BenOrMsgsByzCC$

 $RefinementProperty \triangleq RefMsgsCC!Spec$ \ * Can be uncommented here and in the $BenOrByzCC.cfg$ file to check refinement.
$ConsensusProperty \triangleq \Diamond\Box[ConsensusReachedByzantine]_{vars}$ Eventually, consensus will always be reached.
$CommonCoinProperty \triangleq \Diamond\Box[CommonCoinValid]_{vars}$ Eventually, the $CC$ will be set for all rounds.
---------------------------------------------------------------------------------------------
THEOREM $Spec \Rightarrow \Box\, TypeOK$ Checked by $TLC$.
PROOF OMITTED
---------------------------------------------------------------------------------------------
THEOREM $Spec \Rightarrow ConsensusProperty$ Checked by $TLC$.
PROOF OMITTED
---------------------------------------------------------------------------------------------
THEOREM $Spec \Rightarrow CommonCoinProperty$ Checked by $TLC$.
PROOF OMITTED
=============================================================================================
```

Figure 36. The BenOrMsgsByzCCL TLA^{+} specification

MODULE *BenOrMsgsByzCCLH*

This is a refinement of *BenOrMsgsByzCCL*, *BenOrAbstByzCCL*, and *BenOrAbstByzCCLH* with additional refinement history variables. Several modifications were required for *TLC* to successfully verify the refinement mapping *RefAbst* :

- Added the *estimateHistory* and *proposalHistory* variables that correspond to the *estimatesAtRound* and *proposalsAtRound* variables from *BenOrAbstByzCCL*. All other variables are the same between both specifications and didn't require any additional changes.
- Modified the specification so that whenever a broadcast message is sent, the estimate or proposal value also changes the corresponding history variable.
- Made it so that whenever a node estimates a new value, then both estimates and *estimateHistory* are changed.
- Modified the *Init* formula to also set the *estimateHistory* values based on the initial estimates similarly to *BenOrAbstByzCCL*.
- Defined a new action *SetNextRoundEstimate* that sets the *estimateHistory* values for the next round. This behavior is the same as the *SetNextRoundEstimate* action in *BenOrAbstByzCCL*.
- Added an additional condition for the common coin action that requires for the *estimateHistory* variable to be set when tossing the coin if some node estimated some value. This is required for the refinement mapping *RefAbst*, but doesn't alter the state space in a way that would make the refinement mapping *RefMsgsCCL* incorrect.

The refinement mappings *RefMsgsCCL* and *RefAbstH* are also provided and can be checked with *TLC* if put inside the *RefinementProperty* definition before '!*Spec*'.

EXTENDS *TLC*, *Integers*, *FiniteSets*

Use the *BenOrByzCC.cfg* file.

CONSTANTS *NODES*, *FAULTY_NODES*, *ESTIMATE_ZERO*, *ESTIMATE_ONE*, *CHECK_ALL_INITIAL_VALUES*

VARIABLE *rounds*	Function that returns a given node's current round.
VARIABLE *estimates*	Function that returns a given node's current estimated value.
VARIABLE *decisions*	Function that returns a given node's current decision value.
VARIABLE *msgs*	Set of all sent messages.
VARIABLE *commonCoin*	Models the Common Coin.
VARIABLE *estimateHistory*	History variables for refinement.
VARIABLE *proposalHistory*	

$vars \triangleq \langle rounds, estimates, decisions, msgs, commonCoin, estimateHistory, proposalHistory \rangle$

For the *BenOr Byzantine* consensus protocol version, $N > 5 * F$ must be true. Also, F must be greater than 0, otherwise the $N - F$ check for proposal messages won't work.

ASSUME $\land\ (Cardinality(NODES) > 5 * Cardinality(FAULTY\_NODES))$
$\quad\quad\quad\land\ (Cardinality(FAULTY\_NODES) \geq 1)$

Nodes and their decisions must be specified.

ASSUME $\land\ NODES \neq \{\}$
$\quad\quad\quad\land\ ESTIMATE\_ONE \neq \{\}$
$\quad\quad\quad\land\ ESTIMATE\_ZERO \neq \{\}$

Decision values must be specified for all nodes.

ASSUME $Cardinality(ESTIMATE\_ZERO) + Cardinality(ESTIMATE\_ONE) = Cardinality(NODES)$

Numbers of nodes and faulty nodes.

$NumberOfNodes \triangleq Cardinality(NODES)$
$NumberOfFaultyNodes \triangleq Cardinality(FAULTY\_NODES)$

Set of node round numbers.

$NodeRounds \triangleq Nat$

Set of minimum rounds necessary to reach consensus with the Common Coin.

$CCRounds \triangleq 1\ ..\ 3$

Decision values of binary consensus.

$DecisionValues \triangleq \{\text{“0”}, \text{“1”}\}$

Set of all possible node decision values including the “-1” used for the initial state.

$NodeDecisionValues \triangleq DecisionValues \cup \{\text{“-1”}\}$

Set of all functions that map nodes to their estimate values.

$NodeEstimateFunctions \triangleq [NODES \rightarrow DecisionValues]$

Set of all possible node estimate values. The “-1” is not necessary since all nodes estimate a value of “0” or “1” in the initial state.

$NodeEstimateValues \triangleq DecisionValues$

Set of all functions that map nodes to their estimate values and “-1”.

$AllNodeEstimateFunctions \triangleq [NODES \rightarrow NodeDecisionValues]$

```
Possible values and types of messages in the BenOr consensus algorithm.
MessageType ≜ {"report", "proposal"}
MessageValues ≜ {"0", "1", "?"}

Set of all functions that map nodes to their proposal message values and "-1".
AllNodeProposalFunctions ≜ [NODES → MessageValues ∪ {"-1"}]

Each message record also has a round number.
MessageRecordSet ≜ [Sender : NODES, Round : Nat, Type : MessageType, Value : MessageValues]

Set of all possible Common Coin values. "-1" means that the common coin was not tossed yet for that round.
CommonCoinValues ≜ {"-1", "0", "1"}

Used to print variable values. Must be used with ∧ and in states which are reached by TLC.
PrintVal(message, expression) ≜ Print(⟨message, expression⟩, TRUE)

A set of all nodes other than the specified node.
AllOtherNodes(node) ≜
    NODES \ {node}

Returns TRUE if a node sent the specified message during the specified round.
DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue) ≜
    LET messageRecord ≜ [Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ messageValue]
    IN  messageRecord ∈ msgs

Returns TRUE if a node sent a message of the specified type with any value ("0", "1", or "?") during the specified round.
DidNodeSendMessageOfType(sendingNode, nodeRound, messageType) ≜
    ∃ msgValue ∈ MessageValues :
      LET messageRecord ≜ [Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ msgValue]
      IN  messageRecord ∈ msgs

Adds a specific message to the set msgs with the sending node and the node's round. If the a message with the same value was already sent
during the specified round, it is not added to
 the msgs set.
SendBroadcastMessage(sendingNode, nodeRound, messageType, messageValue) ≜
    ∧ ¬DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue)
    ∧ msgs' = msgs ∪ {[Sender ↦ sendingNode, Round ↦ nodeRound, Type ↦ messageType, Value ↦ messageValue]}
    ∧ IF messageType = "report"
        THEN estimateHistory' = [estimateHistory EXCEPT ![nodeRound][sendingNode] = messageValue]
        ELSE UNCHANGED ⟨estimateHistory⟩
    ∧ IF messageType = "proposal"
        THEN proposalHistory' = [proposalHistory EXCEPT ![nodeRound][sendingNode] = messageValue]
        ELSE UNCHANGED ⟨proposalHistory⟩

Returns the number of messages with the specified value that were sent by nodes other than sendingNode.
HowManyMessages(sendingNode, nodeRound, messageType, messageValue) ≜
    Cardinality({msg ∈ msgs :
                    ∧ msg.Sender ≠ sendingNode
                    ∧ msg.Round = nodeRound
                    ∧ msg.Type = messageType
                    ∧ msg.Value = messageValue})

Returns the number of messages sent by nodes other than checkingNode of the specified message type.
HowManyMessagesOfType(checkingNode, nodeRound, messageType) ≜
    Cardinality({node ∈ AllOtherNodes(checkingNode) :
        DidNodeSendMessageOfType(node, nodeRound, messageType)})

Returns a set of estimate values that are present in more than (N + F)/ 2 report messages. Only one same value can be present in more
than (N + F) / 2 messages.
MajorityOfReports(node, nodeRound) ≜
    {estimateValue ∈ NodeEstimateValues :
        HowManyMessages(node, nodeRound, "report", estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes}

Check if there are more than N / 2 report messages with the same value.
CheckPhase1(node, nodeRound) ≜
```

```
    $MajorityOfReports(node, nodeRound) \neq \{\}$
```

After each node sends a report message during the first phase, it waits for $N - F$ report messages and then counts how many messages contain the same estimate value v.
If there are more than $(N + F)/\ 2$ of such messages, the node sends a proposal message with the value that was in more than $(N + F)/\ 2$ report messages.
Otherwise, the node sends a proposal message with the "?" value. All faulty nodes always send proposal messages with the "?" value.

```
$WaitForReports(node) \triangleq$
    $\land\ node \in NODES$
    $\land\ DidNodeSendMessage(node, rounds[node], \text{"report"}, estimates[node])$
    $\land\ HowManyMessagesOfType(node, rounds[node], \text{"report"}) \geq NumberOfNodes - NumberOfFaultyNodes$
    $\land$ IF $\land\ CheckPhase1(node, rounds[node])$
            $\land\ node \notin FAULTY\_NODES$
        THEN $\exists\, majorityValue \in MajorityOfReports(node, rounds[node]) :$
                $SendBroadcastMessage(node, rounds[node], \text{"proposal"}, majorityValue)$
        ELSE $SendBroadcastMessage(node, rounds[node], \text{"proposal"}, \text{"?"})$
    $\land$ UNCHANGED $\langle rounds, estimates, decisions, commonCoin \rangle$
```

The first phase of the *BenOr* consensus algorithm. All nodes send report messages with their estimate value v to all other nodes. Each node then checks what kind of values they have received from other nodes. The value v that is sent as a proposal message if it is in the majority of report messages
is not necessarily the same as the node's current estimate value. A node can send a value different from its own estimate value to other nodes in a proposal message.

```
$Phase1(node) \triangleq$
    $\land\ node \in NODES$
    $\land\ \lor\ SendBroadcastMessage(node, rounds[node], \text{"report"}, estimates[node])$
       $\lor\ WaitForReports(node)$
    $\land$ UNCHANGED $\langle rounds, estimates, decisions, commonCoin \rangle$
```

Returns a set of proposal values "0" or "1" that are proposed by more than $(N + F)\ /\ 2$ nodes.

```
$MajorityOfProposals(node) \triangleq$
    $\{estimateValue \in NodeEstimateValues :$
        $HowManyMessages(node, rounds[node], \text{"proposal"}, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes\}$
```

Returns a set of proposal values "0" or "1" that are proposed by at least $F + 1$ nodes.

```
$AtLeastNF1Proposals(node) \triangleq$
    $\{estimateValue \in NodeEstimateValues :$
        $HowManyMessages(node, rounds[node], \text{"proposal"}, estimateValue) \geq NumberOfFaultyNodes + 1\}$
```

Check for more than $(N + F)\ /\ 2$ proposal messages with the same value.

```
$CheckPhase2a(node) \triangleq$
    $MajorityOfProposals(node) \neq \{\}$
```

Check for $F + 1$ or more proposal messages with the same value.

```
$CheckPhase2b(node) \triangleq$
    $AtLeastNF1Proposals(node) \neq \{\}$
```

Decide an estimate value that has been proposed by more than $(N + F)\ /\ 2$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$DecidePhase2a(node) \triangleq$
    $\exists\, decisionValue \in MajorityOfProposals(node) :$
        $decisions' = [decisions$ EXCEPT $![node] = decisionValue]$
```

Estimate a proposal value that has been proposed by at least $F + 1$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$EstimatePhase2b(node) \triangleq$
    $\exists\, estimateValue \in AtLeastNF1Proposals(node) :$
        $\land\ estimates' = [estimates$ EXCEPT $![node] = estimateValue]$
        $\land\ estimateHistory' = [estimateHistory$ EXCEPT $![rounds[node]][node] = estimateValue]$
```

Pick a random estimate value from "0" or "1" for a node. *TLC* will check all possible choices separately.

```
$EstimatePhase2c(node) \triangleq$
    $\land\ commonCoin[rounds[node]] \neq \text{"-1"}$
    $\land\ estimates' = [estimates$ EXCEPT $![node] = commonCoin[rounds[node]]]$
    $\land\ estimateHistory' = [estimateHistory$ EXCEPT $![rounds[node]][node] = commonCoin[rounds[node]]]$
```

A node must have sent a proposal message and must wait for $N - F$ proposal messages from other nodes.

```
$WaitForProposals(node) \triangleq$
    $\land\ node \in NODES$
    $\land\ DidNodeSendMessageOfType(node, rounds[node], \text{“proposal”})$
    $\land\ HowManyMessagesOfType(node, rounds[node], \text{“proposal”}) \geq NumberOfNodes - NumberOfFaultyNodes$
```

The second phase of the *BenOr* consensus algorithm. All nodes that have sent a proposal message wait for $N - F$ of such messages from other nodes. If the node receives more than $(N + F)$ / 2 proposal messages with the same value v (“0” or “1”), then
the node sets its decision value to v. Furthermore, if the node receives at least least $F + 1$ proposal messages with the value v, it sets its estimate value to v. Otherwise, the node changes its estimate value to “0” or “1” randomly.

```
$Phase2(node) \triangleq$
    $\land\ node \in NODES$
    $\land\ WaitForProposals(node)$
    $\land$ IF $CheckPhase2a(node) \land CheckPhase2b(node)$
        THEN $DecidePhase2a(node) \land EstimatePhase2b(node)$
        ELSE IF $\lnot CheckPhase2a(node) \land CheckPhase2b(node)$
                THEN $EstimatePhase2b(node) \land$ UNCHANGED $\langle decisions \rangle$
                ELSE IF $\lnot CheckPhase2a(node) \land \lnot CheckPhase2b(node)$
                        THEN $EstimatePhase2c(node) \land$ UNCHANGED $\langle decisions \rangle$
                        ELSE FALSE    No other cases possible.
    $\land\ rounds' = [rounds$ EXCEPT $![node] = @ + 1]$    Node goes to the next round.
    $\land$ UNCHANGED $\langle msgs, commonCoin, proposalHistory \rangle$
```

A node sets the Common Coin value for the current round the node is in. If in the round at least 1 node has already estimated some value non-randomly, then that round's common coin
value will be set to the node's estimate value (the Common Coin was lucky).
Otherwise, the Common Coin value is set to “0” or “1” and *TLC* will check behaviors when all nodes randomly estimate “0” or “1” together.

```
$TossCommonCoin(settingNode) \triangleq$
    $\land\ commonCoin[rounds[settingNode]] = \text{“-1”}$
    $\land\ \lor\ \exists\, node \in NODES \setminus FAULTY\_NODES :$    If at least 1 node already estimated some value in phase 2$b$.
                $\land\ estimateHistory[rounds[settingNode]][node] \neq \text{“-1”}$
                $\land\ WaitForProposals(node)$
                $\land\ \lor\ CheckPhase2a(node) \land CheckPhase2b(node)$
                   $\lor\ \lnot CheckPhase2a(node) \land CheckPhase2b(node)$
                $\land\ commonCoin' = [commonCoin$ EXCEPT $![rounds[settingNode]] = estimates[node]]$
       $\lor\ \land\ \forall\, node \in NODES \setminus FAULTY\_NODES :$
                $\land\ WaitForProposals(node)$
                $\land\ \lnot CheckPhase2a(node)$
                $\land\ \lnot CheckPhase2b(node)$
          $\land\ \exists\, coinValue \in DecisionValues :$    All nodes estimate both values based on the Common Coin.
                $commonCoin' = [commonCoin$ EXCEPT $![rounds[settingNode]] = coinValue]$
    $\land$ UNCHANGED $\langle rounds, estimates, decisions, msgs, estimateHistory, proposalHistory \rangle$
```

Set the node's next round's estimate to the previous round's estimate value so that other nodes are aware that the node has entered the next round. In the previous round the estimate will be some valid value other than “-1”.

```
$SetNextRoundEstimate(node) \triangleq$
    $\land\ node \in NODES$
    $\land\ estimateHistory[rounds[node]][node] = \text{“-1”}$
    $\land\ estimateHistory' = [estimateHistory$ EXCEPT $![rounds[node]][node] = estimateHistory[rounds[node] - 1][node]]$
    $\land\ SendBroadcastMessage(node, rounds[node], \text{“report”}, estimates[node])$
    $\land$ UNCHANGED $\langle rounds, estimates, decisions, commonCoin, proposalHistory \rangle$
```

Consensus property for *TLC* to check. Consensus is guaranteed only between non-faulty nodes. All decision values must be “0” or “1” and all non-faulty nodes must decide
on the same value.

```
$ConsensusReachedByzantine \triangleq$
    $\land\ \forall\, node \in NODES \setminus FAULTY\_NODES : decisions[node] \in DecisionValues$
    $\land\ \forall\, node1, node2 \in NODES \setminus FAULTY\_NODES : decisions[node1] = decisions[node2]$
```

Common Coin validity property for *TLC* to check. The *CC* is valid, if its actually set during every round. Furthermore, it shouldn't take more than 3 rounds to reach consensus with a lucky *CC*.

```
$CommonCoinValid \triangleq$
    $\land\ \forall\, round \in CCRounds : commonCoin[round] \in DecisionValues$
    $\land\ \forall\, node \in NODES : rounds[node] \leq 3$
```

```
The last state of the algorithm when consensus is reached.
LastState ≜
    ∧ ConsensusReachedByzantine

TypeOK ≜
    ∧   ∀ round ∈ CCRounds : estimateHistory[round] ∈ AllNodeEstimateFunctions
    ∧   ∀ round ∈ CCRounds : proposalHistory[round] ∈ AllNodeProposalFunctions
    ∧   ∀ node ∈ NODES : rounds[node] ∈ NodeRounds
    ∧   ∀ node ∈ NODES : estimates[node] ∈ NodeEstimateValues
    ∧   ∀ node ∈ NODES : decisions[node] ∈ NodeDecisionValues
    ∧   msgs ⊆ MessageRecordSet
    ∧   ∀ round ∈ CCRounds : commonCoin[round] ∈ CommonCoinValues

If CHECK_ALL_INITIAL_VALUES is set to TRUE, TLC will check all possible estimate value combinations. Otherwise, initial estimates
 are defined by configuring what nodes estimate a "0" or a "1" value with the sets ESTIMATE_ZERO and ESTIMATE_ONE.
Initial decisions are set to "-1" at the start to ensure ConsensusReached is not true in the initial state.
There are no sent messages in the initial state.
Init ≜
    ∧ IF CHECK_ALL_INITIAL_VALUES
       THEN ∃ estimateFunction ∈ NodeEstimateFunctions :
              ∧ estimateHistory = [round ∈ CCRounds ↦ IF round = 1
                                                         THEN estimateFunction
                                                         ELSE [node ∈ NODES ↦ "-1"]]
              ∧ estimates = estimateFunction
       ELSE   ∧ estimateHistory = [round ∈ CCRounds ↦ IF round = 1
                                                         THEN [node ∈ NODES ↦
                                                            IF node ∈ ESTIMATE_ZERO
                                                             THEN "0"
                                                             ELSE "1"]
                                                         ELSE [node ∈ NODES ↦ "-1"]]
              ∧ estimates = [node ∈ NODES ↦ IF node ∈ ESTIMATE_ZERO THEN "0" ELSE "1"]
    ∧ proposalHistory = [round ∈ CCRounds ↦ [node ∈ NODES ↦ "-1"]]
    ∧ rounds = [node ∈ NODES ↦ 1]
    ∧ decisions = [node ∈ NODES ↦ "-1"]
    ∧ msgs = {}
    ∧ commonCoin = [round ∈ CCRounds ↦ "-1"]

Next ≜
    ∃ node ∈ NODES :
      ∧ ¬LastState  Stops TLC when used with deadlock check enabled.
      ∧ ∨ Phase1(node)
        ∨ Phase2(node)
        ∨ TossCommonCoin(node)
        ∨ SetNextRoundEstimate(node)

Spec ≜
    ∧ Init
    ∧ □[Next]_vars
    ∧ ∀ node ∈ NODES :
        WF_vars(Phase1(node) ∨ Phase2(node) ∨ TossCommonCoin(node) ∨ SetNextRoundEstimate(node))

VERY IMPORTANT: when checking the RefinementProperty, all INSTANCE statements in the other refined history variable specification
 .tla file must be commented out to avoid circular dependencies.
Also, the RefinementProperty definition must be commented out as well since the INSTANCE statements would not be defined in such a case.
RefAbstH ≜
    INSTANCE BenOrAbstByzCCLH
     WITH msgsHistory ← msgs,
          estimateHistory ← estimates,
          estimatesAtRound ← estimateHistory,
          proposalsAtRound ← proposalHistory

RefMsgsCCL ≜
    INSTANCE BenOrMsgsByzCCL
```

$RefAbst \triangleq$
    INSTANCE $BenOrAbstByzCCL$
     WITH $estimatesAtRound \leftarrow estimateHistory,$
          $proposalsAtRound \leftarrow proposalHistory$

$RefinementProperty \triangleq$ Can be uncommented here and in the $BenOrByzCC.cfg$ file to check refinement.
    $\land\ RefAbst!Spec$
    $\land\ RefAbstH!Spec$
    $\land\ RefMsgsCCL!Spec$
$ConsensusProperty \triangleq \Diamond\Box[ConsensusReachedByzantine]_{vars}$ Eventually, consensus will always be reached.
$CommonCoinProperty \triangleq \Diamond\Box[CommonCoinValid]_{vars}$ Eventually, the CC will be set for all rounds.

THEOREM $Spec \Rightarrow \Box TypeOK$ Checked by TLC.
PROOF OMITTED

THEOREM $Spec \Rightarrow ConsensusProperty$ Checked by TLC.
PROOF OMITTED

THEOREM $Spec \Rightarrow CommonCoinProperty$ Checked by TLC.
PROOF OMITTED

Figure 37. The BenOrMsgsByzCCLH TLA^{+} specification

MODULE *BenOrMsgsByzCCLRst*

This specification is a refinement of *BenOrMsgsByzCCL* that allows for only one non-faulty node *CCNode* to toss the common coin. The issue is that if we want to model nodes failing or becoming faulty during the execution of the algorithm, then *CCNode* node must not be compromised or the algorithm won't be able to progress.
Other *BenOrMsgsByz* common coin specifications have their own disadvantages : − In *BenOrMsgsByzCC*, the coin is tossed by the system, no one specific node, yet the state space is a lot greater due to the
coin toss being a system action.
− In *BenOrMsgsByzCCL*, any node can toss the coin, but this leads to a growth of the total number of states as the number of nodes increases.
The refinement mapping *RefMsgsCCL* is also included in the specification and can be checked with *TLC*. The specification contains fewer distinct and total states compared to *BenOrMsgsByzCCL* and as the number
of nodes increase the amount of total states remains lower than *BenOrMsgsByzCCL*.

EXTENDS TLC, $Integers$, $FiniteSets$

Use the *BenOrByzCC.cfg* file.

CONSTANTS $NODES$, $FAULTY\_NODES$, $ESTIMATE\_ZERO$, $ESTIMATE\_ONE$, $CHECK\_ALL\_INITIAL\_VALUES$

VARIABLE $rounds$	Function that returns a given node's current round.
VARIABLE $estimates$	Function that returns a given node's current estimated value.
VARIABLE $decisions$	Function that returns a given node's current decision value.
VARIABLE $msgs$	Set of all sent messages.
VARIABLE $commonCoin$	Models the Common Coin.

$vars \triangleq \langle rounds, estimates, decisions, msgs, commonCoin \rangle$

For the *BenOr Byzantine* consensus protocol version, $N > 5 * F$ must be true. Also, F must be greater than 0, otherwise the $N - F$ check for proposal messages won't work.

ASSUME $\land (Cardinality(NODES) > 5 * Cardinality(FAULTY\_NODES))$
$\quad\quad\;\;\land (Cardinality(FAULTY\_NODES) \geq 1)$

Nodes and their decisions must be specified.

ASSUME $\land NODES \neq \{\}$
$\quad\quad\;\;\land ESTIMATE\_ONE \neq \{\}$
$\quad\quad\;\;\land ESTIMATE\_ZERO \neq \{\}$

Decision values must be specified for all nodes.

ASSUME $Cardinality(ESTIMATE\_ZERO) + Cardinality(ESTIMATE\_ONE) = Cardinality(NODES)$

Numbers of nodes and faulty nodes.

$NumberOfNodes \triangleq Cardinality(NODES)$
$NumberOfFaultyNodes \triangleq Cardinality(FAULTY\_NODES)$

A non-faulty node that sets the common coin.

$CCNode \triangleq$ CHOOSE $node \in NODES : node \notin FAULTY\_NODES$

Set of node round numbers.

$NodeRounds \triangleq Nat$

Set of minimum rounds necessary to reach consensus with the Common Coin.

$CCRounds \triangleq 1 \, .. \, 3$

Decision values of binary consensus.

$DecisionValues \triangleq \{\text{"0"}, \text{"1"}\}$

Set of all possible node decision values including the "-1" used for the initial state.

$NodeDecisionValues \triangleq DecisionValues \cup \{\text{"-1"}\}$

Set of all functions that map nodes to their estimate values.

$NodeEstimateFunctions \triangleq [NODES \rightarrow DecisionValues]$

Set of all possible node estimate values. The "-1" is not necessary since all nodes estimate a value of "0" or "1" in the initial state.

$NodeEstimateValues \triangleq DecisionValues$

Possible values and types of messages in the *BenOr* consensus algorithm.

$MessageType \triangleq \{\text{"report"}, \text{"proposal"}\}$
$MessageValues \triangleq \{\text{"0"}, \text{"1"}, \text{"?"}\}$

Each message record also has a round number.

$MessageRecordSet \triangleq [Sender : NODES, Round : Nat, Type : MessageType, Value : MessageValues]$

Set of all possible Common Coin values. "-1" means that the common coin was not tossed yet for that round.

```
$CommonCoinValues \triangleq \{\text{"-1"}, \text{"0"}, \text{"1"}\}$
```

Used to print variable values. Must be used with $\wedge$ and in states which are reached by *TLC*.

```
$PrintVal(message, expression) \triangleq Print(\langle message, expression \rangle, \text{TRUE})$
```

A set of all nodes other than the specified node.

```
$AllOtherNodes(node) \triangleq$
    $NODES \setminus \{node\}$
```

Returns TRUE if a node sent the specified message during the specified round.

```
$DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue) \triangleq$
    LET $messageRecord \triangleq [Sender \mapsto sendingNode, Round \mapsto nodeRound, Type \mapsto messageType, Value \mapsto messageValue]$
    IN  $messageRecord \in msgs$
```

Returns TRUE if a node sent a message of the specified type with any value ("0", "1", or "?") during the specified round.

```
$DidNodeSendMessageOfType(sendingNode, nodeRound, messageType) \triangleq$
    $\exists msgValue \in MessageValues :$
        LET $messageRecord \triangleq [Sender \mapsto sendingNode, Round \mapsto nodeRound, Type \mapsto messageType, Value \mapsto msgValue]$
        IN  $messageRecord \in msgs$
```

Adds a specific message to the set *msgs* with the sending node and the node's round. If the a message with the same value was already sent during the specified round, it is not added to
the *msgs* set.

```
$SendBroadcastMessage(sendingNode, nodeRound, messageType, messageValue) \triangleq$
    $\wedge \neg DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue)$
    $\wedge msgs' = msgs \cup \{[Sender \mapsto sendingNode, Round \mapsto nodeRound, Type \mapsto messageType, Value \mapsto messageValue]\}$
```

Returns the number of messages with the specified value that were sent by nodes other than *sendingNode*.

```
$HowManyMessages(sendingNode, nodeRound, messageType, messageValue) \triangleq$
    $Cardinality(\{msg \in msgs :$
                    $\wedge msg.Sender \neq sendingNode$
                    $\wedge msg.Round = nodeRound$
                    $\wedge msg.Type = messageType$
                    $\wedge msg.Value = messageValue\})$
```

Returns the number of messages sent by nodes other than *checkingNode* of the specified message type.

```
$HowManyMessagesOfType(checkingNode, nodeRound, messageType) \triangleq$
    $Cardinality(\{node \in AllOtherNodes(checkingNode) :$
        $DidNodeSendMessageOfType(node, nodeRound, messageType)\})$
```

Returns a set of estimate values that are present in more than $(N + F)/\ 2$ report messages. Only one same value can be present in more than $(N + F)\ /\ 2$ messages.

```
$MajorityOfReports(node, nodeRound) \triangleq$
    $\{estimateValue \in NodeEstimateValues :$
        $HowManyMessages(node, nodeRound, \text{"report"}, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes\}$
```

Check if there are more than $N\ /\ 2$ report messages with the same value.

```
$CheckPhase1(node, nodeRound) \triangleq$
    $MajorityOfReports(node, nodeRound) \neq \{\}$
```

After each node sends a report message during the first phase, it waits for $N - F$ report messages and then counts how many messages contain the same estimate value v.
If there are more than $(N + F)/\ 2$ of such messages, the node sends a proposal message with the value that was in more than $(N + F)/\ 2$ report messages.
Otherwise, the node sends a proposal message with the "?" value. All faulty nodes always send proposal messages with the "?" value.

```
$WaitForReports(node) \triangleq$
    $\wedge node \in NODES$
    $\wedge DidNodeSendMessage(node, rounds[node], \text{"report"}, estimates[node])$
    $\wedge HowManyMessagesOfType(node, rounds[node], \text{"report"}) \geq NumberOfNodes - NumberOfFaultyNodes$
    $\wedge$ IF $\wedge CheckPhase1(node, rounds[node])$
          $\wedge node \notin FAULTY\_NODES$
       THEN $\exists majorityValue \in MajorityOfReports(node, rounds[node]) :$
                $SendBroadcastMessage(node, rounds[node], \text{"proposal"}, majorityValue)$
       ELSE $SendBroadcastMessage(node, rounds[node], \text{"proposal"}, \text{"?"})$
    ∧ UNCHANGED ⟨$rounds$, $estimates$, $decisions$, $commonCoin$⟩
```

The first phase of the *BenOr* consensus algorithm. All nodes send report messages with their estimate value v to all other nodes. Each node then checks what kind of values they have received from other nodes. The value v that is sent as a proposal message if it is in the majority of report messages
is not necessarily the same as the node's current estimate value. A node can send a value different from its own estimate value to other nodes in a proposal message.

```
$Phase1(node)$ ≜
    ∧ $node \in NODES$
    ∧ ∨ $SendBroadcastMessage(node, rounds[node],$ "report", $estimates[node])$
      ∨ $WaitForReports(node)$
    ∧ UNCHANGED ⟨$rounds$, $estimates$, $decisions$, $commonCoin$⟩
```

Returns a set of proposal values "0" or "1" that are proposed by more than $(N + F)$ / 2 nodes.

```
$MajorityOfProposals(node)$ ≜
    {$estimateValue \in NodeEstimateValues$ :
        $HowManyMessages(node, rounds[node],$ "proposal", $estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes$}
```

Returns a set of proposal values "0" or "1" that are proposed by at least $F + 1$ nodes.

```
$AtLeastNF1Proposals(node)$ ≜
    {$estimateValue \in NodeEstimateValues$ :
        $HowManyMessages(node, rounds[node],$ "proposal", $estimateValue) \geq NumberOfFaultyNodes + 1$}
```

Check for more than $(N + F)$ / 2 proposal messages with the same value.

```
$CheckPhase2a(node)$ ≜
    $MajorityOfProposals(node) \neq \{\}$
```

Check for $F + 1$ or more proposal messages with the same value.

```
$CheckPhase2b(node)$ ≜
    $AtLeastNF1Proposals(node) \neq \{\}$
```

Decide an estimate value that has been proposed by more than $(N + F)$ / 2 nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$DecidePhase2a(node)$ ≜
    $\exists decisionValue \in MajorityOfProposals(node)$ :
        $decisions' = [decisions$ EXCEPT $![node] = decisionValue]$
```

Estimate a proposal value that has been proposed by at least $F + 1$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$EstimatePhase2b(node)$ ≜
    $\exists estimateValue \in AtLeastNF1Proposals(node)$ :
        $estimates' = [estimates$ EXCEPT $![node] = estimateValue]$
```

Pick a random estimate value from "0" or "1" for a node. *TLC* will check all possible choices separately.

```
$EstimatePhase2c(node)$ ≜
    ∧ $commonCoin[rounds[node]] \neq$ "-1"
    ∧ $estimates' = [estimates$ EXCEPT $![node] = commonCoin[rounds[node]]]$
```

A node must have sent a proposal message and must wait for $N - F$ proposal messages from other nodes.

```
$WaitForProposals(node)$ ≜
    ∧ $node \in NODES$
    ∧ $DidNodeSendMessageOfType(node, rounds[node],$ "proposal"$)$
    ∧ $HowManyMessagesOfType(node, rounds[node],$ "proposal"$) \geq NumberOfNodes - NumberOfFaultyNodes$
```

The second phase of the *BenOr* consensus algorithm. All nodes that have sent a proposal message wait for $N - F$ of such messages from other nodes. If the node receives more than $(N + F)$ / 2 proposal messages with the same value v ("0" or "1"), then
the node sets its decision value to v. Furthermore, if the node receives at least least $F + 1$ proposal messages with the value v, it sets its estimate value to v. Otherwise, the node changes its estimate value to "0" or "1" randomly.

```
$Phase2(node)$ ≜
    ∧ $node \in NODES$
    ∧ $WaitForProposals(node)$
    ∧ IF $CheckPhase2a(node) \land CheckPhase2b(node)$
        THEN $DecidePhase2a(node) \land EstimatePhase2b(node)$
        ELSE IF $\neg CheckPhase2a(node) \land CheckPhase2b(node)$
                THEN $EstimatePhase2b(node)$ ∧ UNCHANGED ⟨$decisions$⟩
                ELSE IF $\neg CheckPhase2a(node) \land \neg CheckPhase2b(node)$
```

```
                    THEN EstimatePhase2c(node) ∧ UNCHANGED ⟨decisions⟩
                    ELSE FALSE    No other cases possible.
    ∧ rounds′ = [rounds EXCEPT ![node] = @ + 1]   Node goes to the next round.
    ∧ UNCHANGED ⟨msgs, commonCoin⟩
```

A node sets the Common Coin value for the current round the node is in. If in the round at least 1 node has already estimated some value non-randomly, then that round's common coin
value will be set to the node's estimate value (the Common Coin was lucky).
Otherwise, the Common Coin value is set to "0" or "1" and *TLC* will check behaviors when all nodes randomly estimate "0" or "1" together.

```
TossCommonCoin(settingNode) ≜
    ∧ settingNode = CCNode
    ∧ commonCoin[rounds[settingNode]] = "-1"
    ∧ ∨ ∃ node ∈ NODES \ FAULTY_NODES :   If at least 1 node already estimated some value in phase 2b.
            ∧ WaitForProposals(node)
            ∧ ∨ CheckPhase2a(node) ∧ CheckPhase2b(node)
              ∨ ¬CheckPhase2a(node) ∧ CheckPhase2b(node)
            ∧ commonCoin′ = [commonCoin EXCEPT ![rounds[settingNode]] = estimates[node]]
      ∨ ∧ ∀ node ∈ NODES \ FAULTY_NODES :
            ∧ WaitForProposals(node)
            ∧ ¬CheckPhase2a(node)
            ∧ ¬CheckPhase2b(node)
        ∧ ∃ coinValue ∈ DecisionValues :   All nodes estimate both values based on the Common Coin.
            commonCoin′ = [commonCoin EXCEPT ![rounds[settingNode]] = coinValue]
    ∧ UNCHANGED ⟨rounds, estimates, decisions, msgs⟩
```

Consensus property for *TLC* to check. Consensus is guaranteed only between non-faulty nodes. All decision values must be "0" or "1" and all non-faulty nodes must decide
on the same value.

```
ConsensusReachedByzantine ≜
    ∧ ∀ node ∈ NODES \ FAULTY_NODES : decisions[node] ∈ DecisionValues
    ∧ ∀ node1, node2 ∈ NODES \ FAULTY_NODES : decisions[node1] = decisions[node2]
```

Common Coin validity property for *TLC* to check. The *CC* is valid, if its actually set during every round. Furthermore, it shouldn't take more than 3 rounds to reach consensus with a lucky *CC*.

```
CommonCoinValid ≜
    ∧ ∀ round ∈ CCRounds : commonCoin[round] ∈ DecisionValues
    ∧ ∀ node ∈ NODES : rounds[node] ≤ 3
```

The last state of the algorithm when consensus is reached.

```
LastState ≜
    ∧ ConsensusReachedByzantine

TypeOK ≜
    ∧   ∀ node ∈ NODES : rounds[node] ∈ NodeRounds
    ∧   ∀ node ∈ NODES : estimates[node] ∈ NodeEstimateValues
    ∧   ∀ node ∈ NODES : decisions[node] ∈ NodeDecisionValues
    ∧   msgs ⊆ MessageRecordSet
    ∧   ∀ round ∈ CCRounds : commonCoin[round] ∈ CommonCoinValues
```

If *CHECK_ALL_INITIAL_VALUES* is set to TRUE, *TLC* will check all possible estimate value combinations. Otherwise, initial estimates are defined by configuring what nodes estimate a "0" or a "1" value with the sets *ESTIMATE_ZERO* and *ESTIMATE_ONE*.
Initial decisions are set to "-1" at the start to ensure *ConsensusReached* is not true in the initial state.
There are no sent messages in the initial state.

```
Init ≜
    ∧ rounds = [node ∈ NODES ↦ 1]
    ∧ IF CHECK_ALL_INITIAL_VALUES
        THEN ∃ estimateFunction ∈ NodeEstimateFunctions : estimates = estimateFunction
        ELSE estimates = [node ∈ NODES ↦ IF node ∈ ESTIMATE_ZERO THEN "0" ELSE "1"]
    ∧ decisions = [node ∈ NODES ↦ "-1"]
    ∧ msgs = {}
    ∧ commonCoin = [round ∈ CCRounds ↦ "-1"]

Next ≜
    ∃ node ∈ NODES :
        $\land \neg LastState$   Stops $TLC$ when used with deadlock check enabled.
        $\land \lor Phase1(node)$
           $\lor Phase2(node)$
           $\lor TossCommonCoin(node)$

$Spec \triangleq$
    $\land Init$
    $\land \Box[Next]_{vars}$
    $\land \forall node \in NODES :$
        $\mathrm{WF}_{vars}(Phase1(node) \lor Phase2(node) \lor TossCommonCoin(node))$

$RefMsgsCCL \triangleq$
    INSTANCE $BenOrMsgsByzCCL$

 $RefinementProperty \triangleq RefMsgsCCL!Spec \setminus *$  Can be uncommented here and in the $BenOrByzCC.cfg$ file to check refinement.
$ConsensusProperty \triangleq \Diamond\Box[ConsensusReachedByzantine]_{vars}$   Eventually, consensus will always be reached.
$CommonCoinProperty \triangleq \Diamond\Box[CommonCoinValid]_{vars}$   Eventually, the $CC$ will be set for all rounds.
```

THEOREM $Spec \Rightarrow \Box TypeOK$ Checked by TLC.
PROOF OMITTED

THEOREM $Spec \Rightarrow ConsensusProperty$ Checked by TLC.
PROOF OMITTED

THEOREM $Spec \Rightarrow CommonCoinProperty$ Checked by TLC.
PROOF OMITTED

Figure 38. The BenOrMsgsByzCCLRst TLA^{+} specification

MODULE *BenOrMsgsByzCCLRstH*

This is a refinement of *BenOrMsgsByzCCLRst*, *BenOrAbstByzCCLRst*, and *BenOrAbstByzCCLRstH* with additional refinement history variables. Several modifications were required for *TLC* to successfully verify the refinement mapping *RefAbst* :

– Added the *estimateHistory* and *proposalHistory* variables that correspond to the *estimatesAtRound* and *proposalsAtRound* variables from *BenOrAbstByzCCLRst*. All other variables are the same between both specifications and didn't require any additional changes.

– Modified the specification so that whenever a broadcast message is sent, the estimate or proposal value also changes the corresponding history variable.

– Made it so that whenever a node estimates a new value, then both estimates and *estimateHistory* are changed.

– Modified the *Init* formula to also set the *estimateHistory* values based on the initial estimates similarly to *BenOrAbstByzCCL*.

– Defined a new action *SetNextRoundEstimate* that sets the *estimateHistory* values for the next round. This behavior is the same as the *SetNextRoundEstimate* action in *BenOrAbstByzCCLRst*.

– Added an additional condition for the common coin action that requires for the *estimateHistory* variable to be set when tossing the coin if some node estimated some value. This is required for the refinement mapping *RefAbst*, but doesn't alter the state space in a way that would make the refinement mapping *RefMsgsCCLRst* incorrect.

The refinement mappings *RefMsgsCCLRst* and *RefAbstH* are also provided and can be checked with *TLC* if put inside the *RefinementProperty* definition before '!*Spec*'.

EXTENDS *TLC*, *Integers*, *FiniteSets*

Use the *BenOrByzCC.cfg* file.

CONSTANTS *NODES*, *FAULTY_NODES*, *ESTIMATE_ZERO*, *ESTIMATE_ONE*, *CHECK_ALL_INITIAL_VALUES*

VARIABLE *rounds* Function that returns a given node's current round.
VARIABLE *estimates* Function that returns a given node's current estimated value.
VARIABLE *decisions* Function that returns a given node's current decision value.
VARIABLE *msgs* Set of all sent messages.
VARIABLE *commonCoin* Models the Common Coin.
VARIABLE *estimateHistory* History variables for refinement.
VARIABLE *proposalHistory*

$vars \triangleq \langle rounds, estimates, decisions, msgs, commonCoin, estimateHistory, proposalHistory \rangle$

For the *BenOr Byzantine* consensus protocol version, $N > 5 * F$ must be true. Also, F must be greater than 0, otherwise the $N - F$ check for proposal messages won't work.

ASSUME $\land\ (Cardinality(NODES) > 5 * Cardinality(FAULTY\_NODES))$
$\quad\quad\quad\land\ (Cardinality(FAULTY\_NODES) \geq 1)$

Nodes and their decisions must be specified.

ASSUME $\land\ NODES \neq \{\}$
$\quad\quad\quad\land\ ESTIMATE\_ONE \neq \{\}$
$\quad\quad\quad\land\ ESTIMATE\_ZERO \neq \{\}$

Decision values must be specified for all nodes.

ASSUME $Cardinality(ESTIMATE\_ZERO) + Cardinality(ESTIMATE\_ONE) = Cardinality(NODES)$

Numbers of nodes and faulty nodes.

$NumberOfNodes \triangleq Cardinality(NODES)$
$NumberOfFaultyNodes \triangleq Cardinality(FAULTY\_NODES)$

A non-faulty node that sets the common coin.

$CCNode \triangleq$ CHOOSE $node \in NODES : node \notin FAULTY\_NODES$

Set of node round numbers.

$NodeRounds \triangleq Nat$

Set of minimum rounds necessary to reach consensus with the Common Coin.

$CCRounds \triangleq 1 \,..\, 3$

Decision values of binary consensus.

$DecisionValues \triangleq \{$"0", "1"$\}$

Set of all possible node decision values including the "-1" used for the initial state.

$NodeDecisionValues \triangleq DecisionValues \cup \{$"-1"$\}$

Set of all functions that map nodes to their estimate values.

$NodeEstimateFunctions \triangleq [NODES \rightarrow DecisionValues]$

Set of all possible node estimate values. The "-1" is not necessary since all nodes estimate a value of "0" or "1" in the initial state.

$NodeEstimateValues \triangleq DecisionValues$

Set of all functions that map nodes to their estimate values and "-1".

$$\mathit{AllNodeEstimateFunctions} \triangleq [\mathit{NODES} \rightarrow \mathit{NodeDecisionValues}]$$

Possible values and types of messages in the *BenOr* consensus algorithm.

$$\mathit{MessageType} \triangleq \{\text{"report"}, \text{"proposal"}\}$$
$$\mathit{MessageValues} \triangleq \{\text{"0"}, \text{"1"}, \text{"?"}\}$$

Set of all functions that map nodes to their proposal message values and "-1".

$$\mathit{AllNodeProposalFunctions} \triangleq [\mathit{NODES} \rightarrow \mathit{MessageValues} \cup \{\text{"-1"}\}]$$

Each message record also has a round number.

$$\mathit{MessageRecordSet} \triangleq [\mathit{Sender} : \mathit{NODES},\ \mathit{Round} : \mathit{Nat},\ \mathit{Type} : \mathit{MessageType},\ \mathit{Value} : \mathit{MessageValues}]$$

Set of all possible Common Coin values. "-1" means that the common coin was not tossed yet for that round.

$$\mathit{CommonCoinValues} \triangleq \{\text{"-1"}, \text{"0"}, \text{"1"}\}$$

Used to print variable values. Must be used with $\land$ and in states which are reached by *TLC*.

$$\mathit{PrintVal}(\mathit{message}, \mathit{expression}) \triangleq \mathit{Print}(\langle \mathit{message}, \mathit{expression}\rangle, \text{TRUE})$$

A set of all nodes other than the specified node.

```
$AllOtherNodes(node) \triangleq$
    $NODES \setminus \{node\}$
```

Returns TRUE if a node sent the specified message during the specified round.

```
$DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue) \triangleq$
    LET $messageRecord \triangleq [Sender \mapsto sendingNode, Round \mapsto nodeRound, Type \mapsto messageType, Value \mapsto messageValue]$
    IN  $messageRecord \in msgs$
```

Returns TRUE if a node sent a message of the specified type with any value ("0", "1", or "?") during the specified round.

```
$DidNodeSendMessageOfType(sendingNode, nodeRound, messageType) \triangleq$
    $\exists msgValue \in MessageValues :$
        LET $messageRecord \triangleq [Sender \mapsto sendingNode, Round \mapsto nodeRound, Type \mapsto messageType, Value \mapsto msgValue]$
        IN  $messageRecord \in msgs$
```

Adds a specific message to the set *msgs* with the sending node and the node's round. If the a message with the same value was already sent during the specified round, it is not added to
the *msgs* set.

```
$SendBroadcastMessage(sendingNode, nodeRound, messageType, messageValue) \triangleq$
    $\land \neg DidNodeSendMessage(sendingNode, nodeRound, messageType, messageValue)$
    $\land msgs' = msgs \cup \{[Sender \mapsto sendingNode, Round \mapsto nodeRound, Type \mapsto messageType, Value \mapsto messageValue]\}$
    $\land$ IF $messageType = $ "report"
        THEN $estimateHistory' = [estimateHistory$ EXCEPT $![nodeRound][sendingNode] = messageValue]$
        ELSE UNCHANGED $\langle estimateHistory \rangle$
    $\land$ IF $messageType = $ "proposal"
        THEN $proposalHistory' = [proposalHistory$ EXCEPT $![nodeRound][sendingNode] = messageValue]$
        ELSE UNCHANGED $\langle proposalHistory \rangle$
```

Returns the number of messages with the specified value that were sent by nodes other than *sendingNode*.

```
$HowManyMessages(sendingNode, nodeRound, messageType, messageValue) \triangleq$
    $Cardinality(\{msg \in msgs :$
                    $\land msg.Sender \neq sendingNode$
                    $\land msg.Round = nodeRound$
                    $\land msg.Type = messageType$
                    $\land msg.Value = messageValue\})$
```

Returns the number of messages sent by nodes other than *checkingNode* of the specified message type.

```
$HowManyMessagesOfType(checkingNode, nodeRound, messageType) \triangleq$
    $Cardinality(\{node \in AllOtherNodes(checkingNode) :$
        $DidNodeSendMessageOfType(node, nodeRound, messageType)\})$
```

Returns a set of estimate values that are present in more than $(N + F)/\ 2$ report messages. Only one same value can be present in more than $(N + F)\ /\ 2$ messages.

```
$MajorityOfReports(node, nodeRound) \triangleq$
    $\{estimateValue \in NodeEstimateValues :$
        $HowManyMessages(node, nodeRound, $ "report" $, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes\}$
```

Check if there are more than N / 2 report messages with the same value.

```
$CheckPhase1(node, nodeRound) \triangleq$
    $MajorityOfReports(node, nodeRound) \neq \{\}$
```

After each node sends a report message during the first phase, it waits for $N - F$ report messages and then counts how many messages contain the same estimate value v.
If there are more than $(N + F)/$ 2 of such messages, the node sends a proposal message with the value that was in more than $(N + F)/$ 2 report messages.
Otherwise, the node sends a proposal message with the "?" value. All faulty nodes always send proposal messages with the "?" value.

```
$WaitForReports(node) \triangleq$
    $\land$ $node \in NODES$
    $\land$ $DidNodeSendMessage(node, rounds[node], \text{"report"}, estimates[node])$
    $\land$ $HowManyMessagesOfType(node, rounds[node], \text{"report"}) \geq NumberOfNodes - NumberOfFaultyNodes$
    $\land$ IF $\land$ $CheckPhase1(node, rounds[node])$
           $\land$ $node \notin FAULTY\_NODES$
       THEN $\exists\, majorityValue \in MajorityOfReports(node, rounds[node])$ :
                 $SendBroadcastMessage(node, rounds[node], \text{"proposal"}, majorityValue)$
       ELSE $SendBroadcastMessage(node, rounds[node], \text{"proposal"}, \text{"?"})$
    $\land$ UNCHANGED $\langle rounds, estimates, decisions, commonCoin \rangle$
```

The first phase of the *BenOr* consensus algorithm. All nodes send report messages with their estimate value v to all other nodes. Each node then checks what kind of values they have received from other nodes. The value v that is sent as a proposal message if it is in the majority of report messages
is not necessarily the same as the node's current estimate value. A node can send a value different from its own estimate value to other nodes in a proposal message.

```
$Phase1(node) \triangleq$
    $\land$ $node \in NODES$
    $\land$ $\lor$ $SendBroadcastMessage(node, rounds[node], \text{"report"}, estimates[node])$
       $\lor$ $WaitForReports(node)$
    $\land$ UNCHANGED $\langle rounds, estimates, decisions, commonCoin \rangle$
```

Returns a set of proposal values "0" or "1" that are proposed by more than $(N + F)$ / 2 nodes.

```
$MajorityOfProposals(node) \triangleq$
    $\{estimateValue \in NodeEstimateValues$ :
        $HowManyMessages(node, rounds[node], \text{"proposal"}, estimateValue) * 2 > NumberOfNodes + NumberOfFaultyNodes\}$
```

Returns a set of proposal values "0" or "1" that are proposed by at least $F + 1$ nodes.

```
$AtLeastNF1Proposals(node) \triangleq$
    $\{estimateValue \in NodeEstimateValues$ :
        $HowManyMessages(node, rounds[node], \text{"proposal"}, estimateValue) \geq NumberOfFaultyNodes + 1\}$
```

Check for more than $(N + F)$ / 2 proposal messages with the same value.

```
$CheckPhase2a(node) \triangleq$
    $MajorityOfProposals(node) \neq \{\}$
```

Check for $F + 1$ or more proposal messages with the same value.

```
$CheckPhase2b(node) \triangleq$
    $AtLeastNF1Proposals(node) \neq \{\}$
```

Decide an estimate value that has been proposed by more than $(N + F)$ / 2 nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$DecidePhase2a(node) \triangleq$
    $\exists\, decisionValue \in MajorityOfProposals(node)$ :
        $decisions' = [decisions$ EXCEPT $![node] = decisionValue]$
```

Estimate a proposal value that has been proposed by at least $F + 1$ nodes. If there are multiple such values, *TLC* will check all possible choices separately.

```
$EstimatePhase2b(node) \triangleq$
    $\exists\, estimateValue \in AtLeastNF1Proposals(node)$ :
        $\land$ $estimates' = [estimates$ EXCEPT $![node] = estimateValue]$
        $\land$ $estimateHistory' = [estimateHistory$ EXCEPT $![rounds[node]][node] = estimateValue]$
```

Pick a random estimate value from "0" or "1" for a node. *TLC* will check all possible choices separately.

```
$EstimatePhase2c(node) \triangleq$
    $\land$ $commonCoin[rounds[node]] \neq \text{"-1"}$
    ∧ estimates′ = [estimates EXCEPT ![node] = commonCoin[rounds[node]]]
    ∧ estimateHistory′ = [estimateHistory EXCEPT ![rounds[node]][node] = commonCoin[rounds[node]]]
```

A node must have sent a proposal message and must wait for $N - F$ proposal messages from other nodes.

```
WaitForProposals(node) ≜
    ∧ node ∈ NODES
    ∧ DidNodeSendMessageOfType(node, rounds[node], "proposal")
    ∧ HowManyMessagesOfType(node, rounds[node], "proposal") ≥ NumberOfNodes − NumberOfFaultyNodes
```

The second phase of the *BenOr* consensus algorithm. All nodes that have sent a proposal message wait for $N - F$ of such messages from other nodes. If the node receives more than $(N + F)$ / 2 proposal messages with the same value v ("0" or "1"), then
the node sets its decision value to v. Furthermore, if the node receives at least least $F + 1$ proposal messages with the value v, it sets its estimate value to v. Otherwise, the node changes its estimate value to "0" or "1" randomly.

```
Phase2(node) ≜
    ∧ node ∈ NODES
    ∧ WaitForProposals(node)
    ∧ IF CheckPhase2a(node) ∧ CheckPhase2b(node)
       THEN DecidePhase2a(node) ∧ EstimatePhase2b(node)
       ELSE IF ¬CheckPhase2a(node) ∧ CheckPhase2b(node)
              THEN EstimatePhase2b(node) ∧ UNCHANGED ⟨decisions⟩
              ELSE IF ¬CheckPhase2a(node) ∧ ¬CheckPhase2b(node)
                     THEN EstimatePhase2c(node) ∧ UNCHANGED ⟨decisions⟩
                     ELSE FALSE   No other cases possible.
    ∧ rounds′ = [rounds EXCEPT ![node] = @ + 1]   Node goes to the next round.
    ∧ UNCHANGED ⟨msgs, commonCoin, proposalHistory⟩
```

CCNode sets the Common Coin value for the current round the node is in. If in the round at least 1 node has already estimated some value non-randomly, then that round's common coin
value will be set to the node's estimate value (the Common Coin was lucky).
Otherwise, the Common Coin value is set to "0" or "1" and *TLC* will check behaviors when all nodes randomly estimate "0" or "1" together.

```
TossCommonCoin(settingNode) ≜
    ∧ settingNode = CCNode
    ∧ commonCoin[rounds[settingNode]] = "-1"
    ∧ ∨ ∃ node ∈ NODES \ FAULTY_NODES :   If at least 1 node already estimated some value in phase 2b.
              ∧ estimateHistory[rounds[settingNode]][node] ≠ "-1"
              ∧ WaitForProposals(node)
              ∧ ∨ CheckPhase2a(node) ∧ CheckPhase2b(node)
                 ∨ ¬CheckPhase2a(node) ∧ CheckPhase2b(node)
              ∧ commonCoin′ = [commonCoin EXCEPT ![rounds[settingNode]] = estimates[node]]
       ∨ ∧ ∀ node ∈ NODES \ FAULTY_NODES :
              ∧ WaitForProposals(node)
              ∧ ¬CheckPhase2a(node)
              ∧ ¬CheckPhase2b(node)
          ∧ ∃ coinValue ∈ DecisionValues :   All nodes estimate both values based on the Common Coin.
              commonCoin′ = [commonCoin EXCEPT ![rounds[settingNode]] = coinValue]
    ∧ UNCHANGED ⟨rounds, estimates, decisions, msgs, estimateHistory, proposalHistory⟩
```

Set the node's next round's estimate to the previous round's estimate value so that other nodes are aware that the node has entered the
next round. In the previous round the estimate will be some valid value other than "-1".

```
SetNextRoundEstimate(node) ≜
    ∧ node ∈ NODES
    ∧ estimateHistory[rounds[node]][node] = "-1"
    ∧ estimateHistory′ = [estimateHistory EXCEPT ![rounds[node]][node] = estimateHistory[rounds[node] − 1][node]]
    ∧ SendBroadcastMessage(node, rounds[node], "report", estimates[node])
    ∧ UNCHANGED ⟨rounds, estimates, decisions, commonCoin, proposalHistory⟩
```

Consensus property for *TLC* to check. Consensus is guaranteed only between non-faulty nodes. All decision values must be "0" or "1" and all non-faulty nodes must decide
on the same value.

```
ConsensusReachedByzantine ≜
    ∧ ∀ node ∈ NODES \ FAULTY_NODES : decisions[node] ∈ DecisionValues
    ∧ ∀ node1, node2 ∈ NODES \ FAULTY_NODES : decisions[node1] = decisions[node2]
```

Common Coin validity property for *TLC* to check. The *CC* is valid, if its actually set during every round. Furthermore, it shouldn't take more than 3 rounds to reach consensus with a lucky *CC*.

```
$CommonCoinValid \triangleq$
    $\land \forall round \in CCRounds : commonCoin[round] \in DecisionValues$
    $\land \forall node \in NODES : rounds[node] \leq 3$
```

The last state of the algorithm when consensus is reached.

```
$LastState \triangleq$
    $\land ConsensusReachedByzantine$

$TypeOK \triangleq$
    $\land \quad \forall round \in CCRounds : estimateHistory[round] \in AllNodeEstimateFunctions$
    $\land \quad \forall round \in CCRounds : proposalHistory[round] \in AllNodeProposalFunctions$
    $\land \quad \forall node \in NODES : rounds[node] \in NodeRounds$
    $\land \quad \forall node \in NODES : estimates[node] \in NodeEstimateValues$
    $\land \quad \forall node \in NODES : decisions[node] \in NodeDecisionValues$
    $\land \quad msgs \subseteq MessageRecordSet$
    $\land \quad \forall round \in CCRounds : commonCoin[round] \in CommonCoinValues$
```

If *CHECK_ALL_INITIAL_VALUES* is set to TRUE, *TLC* will check all possible estimate value combinations. Otherwise, initial estimates are defined by configuring what nodes estimate a "0" or a "1" value with the sets *ESTIMATE_ZERO* and *ESTIMATE_ONE*.
Initial decisions are set to "-1" at the start to ensure *ConsensusReached* is not true in the initial state.
There are no sent messages in the initial state.

```
$Init \triangleq$
    $\land$ IF $CHECK\_ALL\_INITIAL\_VALUES$
       THEN $\exists estimateFunction \in NodeEstimateFunctions$ :
              $\land estimateHistory = [round \in CCRounds \mapsto$ IF $round = 1$
                                                              THEN $estimateFunction$
                                                              ELSE $[node \in NODES \mapsto$ "-1"$]]$
              $\land estimates = estimateFunction$
       ELSE   $\land estimateHistory = [round \in CCRounds \mapsto$ IF $round = 1$
                                                              THEN $[node \in NODES \mapsto$
                                                                 IF $node \in ESTIMATE\_ZERO$
                                                                  THEN "0"
                                                                  ELSE "1"$]$
                                                              ELSE $[node \in NODES \mapsto$ "-1"$]]$
              $\land estimates = [node \in NODES \mapsto$ IF $node \in ESTIMATE\_ZERO$ THEN "0" ELSE "1"$]$
    $\land proposalHistory = [round \in CCRounds \mapsto [node \in NODES \mapsto$ "-1"$]]$
    $\land rounds = [node \in NODES \mapsto 1]$
    $\land decisions = [node \in NODES \mapsto$ "-1"$]$
    $\land msgs = \{\}$
    $\land commonCoin = [round \in CCRounds \mapsto$ "-1"$]$

$Next \triangleq$
    $\exists node \in NODES$ :
       $\land \neg LastState$ Stops TLC when used with deadlock check enabled.
       $\land \lor Phase1(node)$
          $\lor Phase2(node)$
          $\lor TossCommonCoin(node)$
          $\lor SetNextRoundEstimate(node)$

$Spec \triangleq$
    $\land Init$
    $\land \Box[Next]_{vars}$
    $\land \forall node \in NODES$ :
       $WF_{vars}(Phase1(node) \lor Phase2(node) \lor TossCommonCoin(node) \lor SetNextRoundEstimate(node))$
```

VERY IMPORTANT: when checking the *RefinementProperty*, all INSTANCE statements in the other refined history variable specification *.tla* file must be commented out to avoid circular dependencies.
Also, the *RefinementProperty* definition must be commented out as well since the INSTANCE statements would not be defined in such a case.

```
$RefAbstH \triangleq$
   INSTANCE $BenOrAbstByzCCLRstH$
    WITH $msgsHistory \leftarrow msgs$,
         $estimateHistory \leftarrow estimates$,
        estimatesAtRound ← estimateHistory,
        proposalsAtRound ← proposalHistory

RefMsgsCCLRst ≜
    INSTANCE BenOrMsgsByzCCLRst

RefAbst ≜
    INSTANCE BenOrAbstByzCCLRst
     WITH estimatesAtRound ← estimateHistory,
          proposalsAtRound ← proposalHistory

RefinementProperty ≜    Can be uncommented here and in the BenOrByzCC.cfg file to check refinement.
    ∧ RefAbst!Spec
    ∧ RefAbstH!Spec
    ∧ RefMsgsCCLRst!Spec
ConsensusProperty ≜ ◇□[ConsensusReachedByzantine]_vars  Eventually, consensus will always be reached.
CommonCoinProperty ≜ ◇□[CommonCoinValid]_vars  Eventually, the CC will be set for all rounds.
├──────────────────────────────────────────────────────────────────────┤
THEOREM Spec ⇒ □TypeOK  Checked by TLC.
PROOF OMITTED
├──────────────────────────────────────────────────────────────────────┤
THEOREM Spec ⇒ ConsensusProperty  Checked by TLC.
PROOF OMITTED
├──────────────────────────────────────────────────────────────────────┤
THEOREM Spec ⇒ CommonCoinProperty  Checked by TLC.
PROOF OMITTED
└──────────────────────────────────────────────────────────────────────┘
```

Figure 39. The BenOrMsgsByzCCLRstH TLA$^+$ specification