Proving Safety of SupraBFT using Microsoft Ivy

Chandradeep Dey
under supervision of Prof. M Praveen
Chennai Mathematical Institute

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. M Praveen. His expertise and invaluable guidance helped me to complete this research and write this thesis. I am also grateful to my collaborator at AlgoLabs, G Namratha Reddy, for her indispensable contributions to this project.

I would also like to thank the people at SupraOracles, especially Dr. Raghavendra Ramesh, Isaac Doidge, and the research division, for providing me this opportunity to work with their state-of-the-art algorithm.

Finally, I would like to thank my family and friends for their support and encouragement throughout this journey. I could not have done this without them.

Abstract

SupraBFT is an in-development distributed protocol for Byzantine fault-tolerant state machine replication. In this thesis, we attempt to prove certain safety properties for the algorithm. We use Microsoft Ivy as the theorem prover for the task. First, we model the algorithm in the Ivy language. In addition, we write an abstract model that looks at the distributed network from a global perspective. We then prove certain properties about both of the models using inductive invariants.

Contents

1. Background

1.1. Formal methods in protocol verification

Formal methods provide certain guarantees about the nature of communication and security protocols. These guarantees help prevent entire classes of errors from showing up in a faithful implementation of a formally verified protocol.

The method we employ in this article is inductive invariant checking. This boils down to checking 1. whether the properties hold in the initial state of the system, and 2. starting from a state where the properties hold, whether the properties hold in the next state of the system.

Now, the properties we specify need to be expressed in some logic, and the undecidability of first-order logic is a well-known result[6][5].

1.2. A positive answer to the Entscheidungsproblem

1.2.1. The EPR fragment

As such, program verification often focuses on decidable fragments of the logic. A notable such fragment is the Bernays-Schönfinkel-Ramsey class. Also called the *effectively propositional reasoning* (EPR) fragment, formulæ in this class have the prefix $\exists^*\forall^*$ in prenex normal form[1], and have no function symbols. The EPR fragment is decidable[12], and a decision procedure may be found in [11, Section 9.4].

1.2.2. Extending EPR

EPR can be extended to get back a good fraction of first order logic while preserving decidability[8]. We will now build up to the *finite almost uninterpreted* (FAU) fragment.

First, we allow multiple sorts instead of restricting ourselves to a single sort. Now we introduce function symbols back. However, we make sure that the functions are stratified. We can check for this algorithmically by checking for cycles in a graph consisting of the sorts as the vertices and the function symbols as directed edges.

Turns out, we can even have some quantifier alterations. For the decision procedure, we replace all existential quantifications by Skolem functions. Therefore, we just make sure to accept only those formulæ that do not have the cycles post-Skolemisation.

Next up, we consider what is called the relevant vocabularies[8] for every universally quantified variable, uninterpreted function symbol, and uninterpreted relation symbol. Let $V[X]$ be the relevant vocabulary of the term X. We use the following rules—

[1] Quantifiers and bound variables followed by a quantifier-free formula

1. If $t(x_1, \cdots, x_n)$ is the ith argument of f, every instance $t(V[x_1], \cdots, V[x_n])$ is in $V[f, i]$.
2. If X is universally quantified, and it is the ith argument of f, then $V[X] = V[f, i]$.
3. For a type `u`, for the relation $X : \texttt{u} = e$ or $e = X : \texttt{u}$, we define $V[X] = V[u]$.
4. For a type `u`, for the relation $t(x_1, \cdots, x_n) : \texttt{u} = e$ or $e = t(x_1, \cdots, x_n) : \texttt{u}$, we say that every instance $t(V[x_1], \cdots, V[x_n])$ is in $V[u]$.

However, if we start computing the relevant vocabularies, we might never reach a fixpoint and so, the algorithm might never terminate. Luckily, we can once again do a cycle detection[14].

With finite relevant vocabularies and universally quantified variables only as arguments of uninterpreted function or relation symbols, we have what is called the *finite essentially uninterpreted* (FEU) fragment.

Finally, we go one step further by allowing universally quantified variables to occur as arguments to arithmetic literals[1]. This is the FAU fragment.

1.3. Microsoft Ivy

Microsoft Ivy[17][15][16] is both a programming language with Hoare-style property checking and a proof assistant. Its distinguishing feature is a fragment checker that forces the verification conditions to be in the FAU fragment and Z3 as a decision procedure for the fragment, allowing automated checking of the verification conditions. What follows is a description of a subset of the Ivy syntax that we will be using in our code.

Ivy files have the extension `.ivy`. One file may `include` other files. This inclusion behaves somewhat like the `#include` directive of a C/C++ preprocessor, which create 'translation unit's.

Now, a processed Ivy 'unit' consists of `isolate`s, units of verification that can be checked independently. There is an implicit outermost isolate. Isolates can contain other isolates. They can treat properties and invariants proven in external isolates as `axiom`s. For this, a `with <comma separated isolate names>` syntax is used. However, anything that is inside a `private` block has no effect outside the isolate. The current isolate (which may be the implicit top-level one) can be referred to using the `this` keyword. Isolates have other properties, which we will come to after discussing `object`s.

We move on to objects. The `module` keyword groups a bunch of declaration, procedures, and invariants together. They have to be instantiated to create objects. The `instance` keyword lets us create the objects, while the `instantiate` keyword dumps the contents of the module into the current scope after appropriate substitutions for the module parameters. A module with only a single instance can be created with the `object` keyword. We typically parameterise objects, such as `object node (pid:pid_t)`. In this case, `node` acts more like a class, whereas `node(0)`, `node(1)`, ... behave like the objects. Now we go on to another tangent, about the `type` keyword.

[1] predicates or their negations that are interpreted using the rules of arithmetic

The `type` keyword lets us specify new uninterpreted sorts. We can then create variables of these types using either the `individual` or the `var` keyword. The `individual` keyword creates a variable that can be accessed as a member of the object, whereas `var` is used inside procedures for variables that exist only in the scope of that procedure. We can use the `interpret` keyword to give interpretations to types (such as `nat` or bit vectors), but that might lead us out of the decidable fragment.

Back to objects—we can write `type this` inside an object to create a new type that has the same name as the object. The functions, relations, `individuals`, and invariants that the object contained become associated with the type. This lets us define types with interesting properties without having to interpret them fully. Note that this does not work with parameterised objects.

Now we come back to isolates—isolates also generate objects. Instead of writing

```
isolate foo = {
    object foo = {
        ...
    }
}
```

we just write

```
isolate foo = {
    ...
}
```

Though in the first case we would have to use `foo.foo` to access the actual object (`.` is the membership operator).

Finally, procedures are defined using the `action` keyword. An `action` may be `export`-ed. Inductiveness of invariants is checked for arbitrary calls (in arbitrary order) to exported actions. Actions that are not exported are really only used for managing code. All actions support Hoare-style pre-conditions (`requires`) and post-conditions (`ensures`) that are checked by Z3. We can choose to split up an action into multiple parts using the `before` keyword for the pre-conditions, `implement` keyword for the action body, and the `after` keyword for the post-conditions.

1.4. Consensus protocols

A principal component underlying blockchain systems is a consensus protocol. Consensus protocols are Byzantine fault-tolerant (BFT) distributed protocols which provide state machine replication (SMR). An SMR protocol replicates the order of operations (and thus, the state of the underlying state machine) across multiple nodes of the system. A BFT-SMR protocol guarantees replication in the presence of a malicious adversary, which controls up to a certain fraction of the total nodes. The bound on the number of such adversaries for such protocols is a classical result[1].

There are many consensus protocols in literature[4][2][18][9] along with attempts at proving them[3][13][10]. We prove properties about a novel BFT-SMR protocol, Supra-BFT.

An instance of the PBFT[4] protocol has three phases—propose[1], prepare, and commit. Each round has a leader. Once a client makes a request for some computation, the leader sends a 'proposal message' to all the nodes in the network. Once the nodes receive the message, they send out 'prepare message's. Any node that receives at least $2f + 1$ (where f is the maximum number of Byzantine nodes the network can handle) prepare messages produces a prepare certificate and broadcasts it out. Finally, nodes that receive $2f + 1$ prepare certificates perform the computation corresponding to the request and send out the result, as a response, to the client. In addition, they generate commit certificates as guarantees of their receipt of the prepare certificates. There are mechanisms for the nodes to time out when they do not receive a message properly, and for bringing a node, that might have timed out, up to date.

PBFT, and BFT-SMR protocols in general, guarantee linearisability of the requests and responses log. Now, we can replace the client request and response parts with functions that create new blocks and add them to the blockchain. Tendermint[2] is such an implementation of PBFT for usage in blockchain systems.

The HotStuff paper[18] introduces a fundamental improvement to protocols like PBFT and Tendermint through chaining. Once a prepare certificate is formed, it is sent to the leader of the next round. In the Chained HotStuff protocol, the propose[2] phase of the next round serves as the commit[3] phase of the earlier round, as the next proposal uses the aforementioned prepare certificate.

A distinguishing feature of SupraBFT, compared to existing protocols, is that it tries to chain results of multiple rounds of computation without waiting for the previous rounds to finish completely. In particular, in addition to the propose phase of the next block serving as the prepare phase of the current block, it also serves as the commit phase of the previous block.

The properties we are interested in are safety (local blockchains across two honest nodes are either the same, or one is a subchain of the other) and liveness (new blocks keep getting added to the blockchain). These guarantees are, of course, only possible up to certain network assumptions. There are classical results in distributed computing which state that it is impossible to arrive at such a consensus in an asynchronous network[7].

However, this is mostly relevant for liveness. Our work in this article is solely the safety proof, and proving that the protocol is safe in an asynchronous network is simply a stronger claim than proving that it is safe in partially synchronous networks.

[1] 'Pre-prepare' in the original paper.
[2] 'Prepare' in the original paper.
[3] 'Decide' in the original paper.

2. SupraBFT

2.1. Outline of the Ivy code

2.1.1. `ubd_seq.ivy`

This file copies the `unbounded_sequence` module from the `order` standard library. The name `unbounded_sequence` should be self-explanatory. The difference between `unbounded_sequence` and `ubd_seq` is that the successor relation is a definition rather than an implication. With `unbounded_sequence`, we cannot use the successor relation as an antecedent in invariants because, the successor relation may be set to true even though its requirements are false.

2.1.2. `domain_model.ivy`

This file defines all the types. `round_t` and `height_t` are unbounded sequences. The protocol proceeds in rounds, and every node has various round counters. `height_t` is used for the height of the block in the blockchain.

`node_t` is an `iterable`. We can iterate through the nodes using the iterator provided by its interface.

`hash_t` is an uninterpreted type representing the block hash. A specific hash is given the name `nil`.

`block_t` defines the blocks. The `consBlock` action can be called to construct new blocks. There is a special `nil` block. We make sure that, and that only, the `nil` block has the `nil` parent hash. Note that the hash of the `nil` block is not `nil`.

`quorum_t` defines a quorum (set) of nodes. We make sure that every quorum has a good member[1], and that the intersection of any two quorums has a good node. These axioms are hidden from the rest of the file, though, as they can easily lead us out of the decidable fragment.

`voted_t` is just a wrapper around a block.

`cert_t` objects consists of a block and a quorum whose members have all voted on the block. This is enforced using requirements in the actions where `cert_t` objects are parameters.

`timeout_t` contains the round number and the highest certificate that a node had when it sent a time-out message for that round.

`timeout_cert_t` collects `timeout_t` the same way `cert_t` collects `voted_t`.

`proposal_t` contains the things required for a new block to be proposed—the block, either a quorum certificate for the grandparent or an appropriate time-out certificate.

[1] Assuming f faulty nodes, $3f + 1$ total nodes, and $2f + 1$ nodes in each quorum.

2.1.3. `network_model.ivy`

This file implements the network. It is mostly a copy from the Tendermint proof[13], adapted for our message types. We set a `sent` relation whenever a message is sent, and require that the relation be set for the message to be received. This lets us drop packets[1], repeat packets, or arbitrarily delay packets.

2.1.4. `algorithm.ivy`

`algorithm.ivy` contains an implementation of the algorithm that a node in the network follows.

After the list of local variables and the initialisation block, we have the network message handlers. These set the appropriate 'message received' relations to true for correct messages. The rest of the algorithm only works with messages that have the received relation set to true.

Then there are the network broadcast handlers. These take various objects of various types, package them into appropriate network packets, and broadcast them on the network.

The aggregation functions aggregate prepare/time-out messages and create `cert_t` or `timeout_cert_t` objects for which the received relation is set to true. It should be noted that the certificates might directly be received from the network. This is handled by the handlers above.

Time-out is simulated through an exported action. The environment can call this action arbitrarily to set the `timed_out` boolean. The boolean is used to control some of the exported actions through pre-conditions on its value.

The `byzantine_send` action simulates actions taken by a Byzantine process. In short, it can send any message that is valid.

`advanceToRound` moves the protocol to a new round. We vote for blocks, and, if we are the leader, propose new blocks here. We also reset the round timer by setting the `timed_out` variable to false. For a new block, the parent depends on whether the last block was voted on or whether the node received a timeout certificate, in which case the highest block from the timeout certificate is chosen (`locked` contains the current node's highest block).

In `tryLockAndCommit`, we check if, with the input quorum certificate (QC), we have two successive blocks in our uncommitted storage. In that case, we commit the older one and all of its uncommitted ancestors. In addition, we also update the locked variable if the QC is newer.

`process_prepareQC` is our first exported action. It is called by the environment with an arbitrary QC. We use the `require` condition to restrict the possible states to those where the environment calls it with received certificates only. What we do here is store the QC in a forest of certificates corresponding to uncommitted blocks. We call `tryLockAndCommit` with the QC to see if the new block can be locked or committed. The

[1] Runs of the protocol where the environment never calls the action, that receives the packet, with the specific message.

reason it might not be lockable is that it might be a much older block whose certificate got delayed.

`process_proposal` reacts to receiving a proposal message. We check if it is for the correct round, whether the hashes are correct, and so on. We then call `advanceToRound` for voting and (maybe) proposing the next block.

`timeout` is used to broadcast a time-out message when the node times out.

`fallback` relates the abstract `timed_out` variable with the time-out code from the real algorithm.

In `timeout_synchronisation`, the current node is forced to time out when it sees a time-out message from at least one honest node.

`process_timeoutQC` reacts to receiving a time-out certificate (TC). `advanceToRound` is called with the TC contents for further processing.

`fallback_recovery` is similar to `process_proposal`, but here we process a proposal that was constructed based on a TC (a fallback-recovery proposal).

`finished_prepareQC_processing` does some additional housekeeping to let proposals and votes go through smoothly for QCs received in weird orders. `no_proposal` and `leader_failed` have a similar job, they let a leader propose the next block based on the locked block with or without voting for a block in `advanceToRound`.

`proposal_b0_correct` deals with the genesis block, which has a nil parent hash, and needs to be committed without much verification.

`proposal_b0_late` deals with the situation where there is a timeout in the first round.

`proposal_b1_correct` deals with the next block. It also needs a special treatment because there is no grandparent.

`b0_qc` deals with processing the genesis QC, we mostly admit it with minimal checks about its structure.

Now we have a bunch of invariants that prove some properties about these actions. We also make sure that certain things in the global view and this file happen together. What follows is an English description of the invariants. We will use the words block and QC somewhat interchangeably.

1. If `rcvd_prepare` is true here, then for that block and that node, `node_has_voted` should be true in the global view.
2. Same for time-out messages.
3. For honest nodes, its current round is always greater than the round of whatever block it voted for.
4. Only valid QCs are received (others get ignored through the various `require` conditions).
5. Only received QCs can be processed.
6. Only valid TCs are received.
7. If a proposal messages is marked as received, whatever certificates it may contain must also be marked as received.

8. The `nil` proposal is never received.
9. The `nil` prepare message is never received.
10. The `nil` time-out message is never received.
11. The `nil` QC is never received.
12. The `nil` TC is never received.
13. For honest nodes, its current round is always greater than the round of whatever block is currently locked.
14. Locking here corresponds to setting the `node_has_locked_detail` relation in the global view with the appropriate round number.
15. Locking here corresponds to setting the `node_has_locked` relation in the global view.
16. If locked is empty or contains the genesis block, then nothing has been committed yet.
17. The blockchain is well-formed (there are no blocks beyond `chain_size`).
18. The blockchain is well-formed (blocks are filled upto `chain_size`).
19. For every block on the blockchain, we have processed a QC related to that block.
20. The currently locked block is in the uncommitted forest.
21. Only correct QCs are in the uncommitted forest, and the ancestor relation for blocks inside the forest is reflexive.
22. The `forest_ancestor` relation is set only for blocks stored in the forest.
23. A block cannot simultaneously be in the uncommitted forest and the blockchain.
24. If there is nothing in the blockchain, then the forest root is the genesis block.
25. If the block in the forest root has height zero, then there is nothing in the chain and it is the genesis block.
26. Blocks in the forest have height greater than heights in the chain.
27. If there is a block in the forest with height one more than the maximum in the chain, then it is the root of the forest.
28. If there is a forest root, then its parent is the last block in the blockchain.
29. If `forest_ancestor` is set for two blocks with consecutive heights, then one is the parent of the other.

30. If two blocks are stored in the forest and the `forest_succ` relation is set for them, then one is the parent of the other and have successive heights.
31. Blocks in the blockchain respect the parent relation.
32. `forest_ancestor` being set implies `forest_store` being set.
33. Only one QC corresponding to a block is stored in the forest.
34. The `forest_strictAncestor` relation is not reflexive.
35. Only one QC is stored in the forest for a particular height.
36. `forest_ancestor` is antisymmetric.
37. `forest_ancestor` is transitive.
38. If two blocks have a common ancestor, then one is the ancestor of the other.

2.1.5. `global_view.ivy`

As the name suggests, the global view tracks the progression of the algorithm from a global perspective. The actions here are not part of the algorithm, but they are used in the proof to reason about things that happen across multiple nodes. Things we track include when a node votes, when it locks something, when it processes a QC, and so on.

2.2. Safety proof using inductive invariants

The proofs primarily revolve around the relations `gdc` and `ldc`, which talk about nodes (or sets of nodes) locking blocks. However, a lot of the invariants try to relate the global view and the local view of the algorithm. The proof is split into multiple isolates. Some of them depend on other isolates, treating the properties in them as axioms. Some of the invariants are duplicated to make the particular isolate run faster, which means we need additional invariants relating similar relations. Towards the end, there are proofs that involve induction. This is done by definining custom induction tactics over `round_t` and `height_t`. The main proofs are towards the end, in the `global_properties` isolate.

2.2.1. `classic_safety.ivy`

`isolate gv = {` This isolate instantiates the `global_view` module and proves some properties about the module, similar to the invariants in `algorithm.ivy`.

1. If `node_has_voted_detail` is set for a block, then it points to the correct round and parent, and the parent has been voted.
2. If `node_has_locked` is set for a block, there is a quorum of nodes who have all voted for the block.

3. An honest node votes for only one block in a round.

4. Two honest nodes cannot lock two different blocks with the same round.

5. If `node_has_processed_qc` is set for some QC, then there is a quorum of nodes who have voted for the block.

6. If two nodes have processed two QCs with the same round number, then they are for the same block.

7. If a node has voted for the genesis block, then there is no block that is locked with a round less than the genesis block.

8. QCs are never processed for the `nil` block.

}
`isolate gdc_properties = {` The `gdc` relation is true for blocks that have a quorum of nodes who have locked the block in the round next to the block's round.

1. A block's parent has a lower round than the block.

2. The `nil` block has the lowest round.

}
`isolate gdc_properties1 = {`

1. If `node_has_voted_detail` is set, then the round number and parent hash is correct, `node_has_voted` is set, and either the parent is `nil` (the block is genesis), or the parent was the most recently locked block within that round.

2. A block is locked in a round later than the round it was constructed in.

3. If `node_has_voted` is set, then there is a appropriate `node_has_voted_detail` set as well.

4. The `quorum_of_votes` relation is well-defined with respect to a quorum of nodes for whom the `node_has_voted_detail` relation is appropriately set.

5. If a non-genesis block is locked, `quorum_of_votes` should be true with the appropriate parent.

6. If a block is locked, `quorum_of_votes` should be true with the appropriate parent.

7. If `quorum_of_recent_locks` is set for a block and some round, then there is a quorum of nodes whose most recent locked block within that round is that block.

}
`isolate gdc_properties2 = {`

1. If `node_has_voted_detail` is set, then the round number and parent hash is correct, `node_has_voted` is set, and either the parent is `nil` (the block is genesis), or the parent was the most recently locked block within that round.

2. If `node_has_voted_detail` is set for a block with parent not `nil`, then the round number and parent hash is correct, `node_has_voted` is set, and the parent was the most recently locked block within that round.

3. In `round_t`, `succ` is unique for a round.

4. If `quorum_of_votes` is set, then there is a quorum whose good members have a corresponding `node_has_voted_detail` set.

5. If there is a quorum of nodes who voted for a block, and we know that if a block was voted for by a node, then its parent was locked recently, then there is a quorum of nodes who locked the parent recently.

6. If there `quorum_of_votes` is set for some block for some parent, then there is a quorum of locks for the parent.

```
}
isolate gdc_properties2a = {
```

1. If `quorum_of_votes` is set for some block, then `quorum_of_recent_locks` will be set for the parent.

2. Nodes can only lock blocks in rounds higher than the round of the block.

3. `node_has_locked_recently` states which block was the most recently locked block for a particular round. It should correspond to `node_has_locked_detail` for that round.

4. If a block is locked, then the GDC relation became true for its parent (it got a quorum of locks).

```
}
isolate gdc_properties3 = {
```

1. In `round_t`, `succ` is unique for a round.

2. If `node_has_voted` is set, then there is a appropriate `node_has_voted_detail` set as well.

3. If `node_has_processed_qc` is set for some QC, then there is a quorum of nodes who have voted for the block.

4. If `node_has_processed_qc` is set for some QC, then there is a quorum of nodes who have voted for the block, but this time we talk about the `node_has_voted_detail` relation instead.

5. If `node_has_voted_detail` is set, then the round number and parent hash is correct, `node_has_voted` is set, and either the parent is `nil` (the block is genesis), or the parent was the most recently locked block within that round.

6. If `node_has_processed_qc` is set for some QC, then there is a quorum of nodes who have voted for the block. This quorum corresponds exactly to the `cert` member of the `cert_t` individual representing the QC.

}
isolate gdc_properties4 = {

1. If a block is marked `gdc`, then the `node_has_locked_detail` relation is set appropriately for a quorum of nodes.

2. If a block is marked `gdc`, then the `node_has_locked_detail` relation is set appropriately for some node.

}
`isolate gdc_properties5 = {` The `quorum_of_locks` relation is true for blocks that have a quorum of nodes for which `node_has_locked_detail` is true.

1. If something has the `quorum_of_recent_locks` relation true, then there is a set of nodes for which `node_has_locked_recently` is true.

2. The quorum of locks relation is true for blocks that have a quorum of nodes for which `node_has_locked_detail` is true.

3. Any `quorum_of_locks` and `quorum_of_recent_locks` share a node that has done `node_has_locked_detail` for the former and `node_has_locked_recently` for the latter.

4. Same thing as above, but we now require a ordering for the two blocks to get a stronger statement.

}
`isolate gdc_supplementary_defs = {` The `no_lock` relation is true for blocks that do not have a lock. `node_has_two_locks` is true if given three blocks, the middle one and the last one are locked, the last one was locked recently, and the parent of the first one is the third one. `node_has_recent_lock` is stronger, it claims `no_lock` for blocks beyond Bp.
}
isolate gdc_properties6 = {

1. If `node_has_locked_recently` is true for some block, then it does not have a lock for some lower round.

2. For two blocks with locks, there is a round lower than the lower round block, where the higher round block was not locked.

3. If node_has_two_locks is true, so is node_has_recent_lock.

}
isolate gdc_properties7m1 = {

1. node_has_locked implies having a quorum_of_votes.

}
isolate gdc_properties7m2 = {

1. quorum_of_votes implies quorum_of_recent_locks for the parent.

}
isolate gdc_properties7 = {

1. If node_has_locked is true for some block, then when the block got locked, its parent got a quorum_of_recent_locks.

}
isolate gdc_properties8 = {

1. If a block is marked gdc, then quorum_of_locks is true for it.

}
isolate gdc_properties9 = {

1. If some block is locked, and some older block is gdc, there is a quorum_of_locks for the gdc block and a quorum_of_recent_locks for the parent block of the locked block.

}
isolate gdc_properties10 = { The gdc_and_lock relation is true for a block that is locked and an older block that has a quorum of locks (gdc).

1. If gdc_and_lock is true, then node_has_two_locks is also true for similar reasons.

}
isolate gdc_properties11 = {

1. gdc_and_lock implies node_has_recent_lock.

}
gdc_properties9am1 = {

1. quorum_of_votes implies quorum_of_recent_locks for the parent.

}
isolate gdc_properties9a = {

1. If some block is locked, and some older block is gdc, there is a quorum_of_locks for the gdc block and a quorum_of_recent_locks for the parent block of the locked block.

2. Similar, but asserts existence of the rounds instead.

3. Similar, removes `is_good(N)` from the premises.

}

`gdc_properties10a = {` `gdc_and_quorum` refers to a block having a quorum of votes, and an older block being `gdc`.

1. `gdc_and_quorum` implies `node_has_two_locks`.

}

`isolate gdc_properties11a = {`

1. `gdc_and_quorum` implies `node_has_recent_lock`.

}

`isolate quorum_propagationm1 = {`

1. `quorum_of_votes` implies `quorum_of_recent_locks` for the parent.

}

`isolate quorum_propagationm2 = {`

1. If `node_has_voted` is set, then there is a appropriate `node_has_voted_detail` set as well.

2. A block's parent has a lower round than the block.

3. The `nil` block has the lowest round.

4. The `quorum_of_votes` relation is well-defined with respect to a quorum of nodes for whom the `node_has_voted_detail` relation is appropriately set.

5. A block has a unique parent.

6. `node_has_locked_detail` corresponds to `node_has_locked`.

7. `node_has_locked` implies `quorum_of_votes` for the block.

}

`isolate quorum_propagation = {` `block_has_qov` lets us specify a block with a `quorum_of_votes`, without mentioning the parent explicitly.

1. `quorum_of_recent_locks` implies that there is a `node_has_locked`.

2. `node_has_voted` implies that the block has a parent.

3. If a block has `block_has_qov`, then so does its parent.

}

`isolate gdc_properties12 = {` `nil_locks_upto_round` is true if given a round, the only blocks marked `node_has_locked_detail` are `nil` blocks.

1. `block_has_qov` implies `quorum_of_votes`.
2. If the genesis block has a `quorum_of_votes`, then for all rounds before that, there is only `nil_locks_upto_round`.
3. Similar, but for `block_has_qov` rather than `quorum_of_votes`.

}
isolate gdc_properties13 = {

1. There is one honest node between a quorum of nodes that have only `nil` locked till a particular round and a quorum of nodes that have locked a block.
2. The conclusion of this statement is its premises.
3. In case something like the first one here happens, it must be the case that the gdc block has a higher round.
4. Similar for the second case.

}
isolate gdc_properties14 = {

1. If the genesis block has `block_has_qov` and some other block is `gdc`, then that block has a higher round.

}
isolate gdc_properties15 = {

1. If a non-genesis block is `gdc`, then there is some node that has locked it, and the block has `quorum_of_votes`.
2. If any block is `gdc`, then there is some node that has locked it, and the block has `quorum_of_votes`.

}
isolate gdc_properties16 = {

1. `quorum_of_votes` implies `block_has_qov`.
2. If any block is `gdc`, then there is some node that has locked it, and the block has `quorum_of_votes`.
3. If any block is `gdc`, then `block_has_qov`.

}
isolate basic_safety = {
isolate processor = { This isolate instantiates the algorithm with the global view instantiation as an argument.

1. A block's parent has a lower round than the block.
2. The `nil` block has the lowest round.
3. The `sent` relation in the network model correspons to a global view relation.
4. Same.
5. If there is a block in the blockchain, then there is a processed QC for that block.
6. The `node_has_locked_detail` relation from the global view corresponds to locking here.
7. Similar.
8. The current round of a node is always greater than whatever it has voted for.
9. The current round of a node is always greater than whatever it has locked.
10. `qc_processed` of the local view is related to `node_has_processed_qc` from the global view.

`}`
`isolate continuity = {`

1. If a block has some ancestor in the forest at some height, then it has ancestors in inbetween heights.
2. Some basic property about height.

`}`
`isolate ldc_propertiesm1 = {` The `ldc` relation refers to a block being locked by some node. The `gdc_x` relation is the same as the `gdc` relation from before.
`}`
`isoalte ldc_propertiesm2 = {`

1. `ldc` implies `gdc` of the parent.

`}`
`isolate ldc_properties = {`

1. The current round of a node is always greater than whatever it has locked.
2. `node_has_locked_detail` corresponds to the node's internal `qc_processed` relation being set.
3. If two blocks are locked, then their rounds have the same order as the rounds they were locked in.
4. Bunch of things about the block at the tip of the blockchain.

5. The `locked` variable corresponds to a leaf in the forest at the appropriate height.
6. The block to be committed was locked.
7. Equality of two individuals inside the node objects.
8. If the blockchain is non-empty, then `commit_candidates` is set.
9. Some more properties about the tip of the blockchain.
10. The `dc_basis_block` is `ldc`.

}
`isolate ldc_properties1 = {`

1. The tip of the chain is `ldc`.

}
`isolate ldc_properties2 = {`

1. The tip of the chain is `gdc_x`.
2. If `node_has_recent_lock` is true, then the middle argument block is older.

}
`isolate ldc_properties2a = {`

1. If `node_has_recent_lock` is true, then the middle argument block is older.

}
`isolate ldc_properties3 = {`

1. `gdc_and_lock` is related to `node_has_recent_lock`.

}
`isolate ldc_properties4 = {`

1. `node_has_locked` and `gdc` combined gives us `gdc_and_lock`.

}
`isolate ldc_properties5 = {`

1. `node_has_locked` and `gdc_x` combined gives us `node_has_recent_lock`.

}
`isolate ldc_properties6 = {`

1. `node_has_locked` and `gdc_x` for an older block combined give us that the older block's round is lower than the parent of the locked block.

}
`isolate ldc_properties3a = {`

1. gdc_and_quorum implies node_has_recent_lock.

}
isolate ldc_properties4a = {

1. quorum_of_votes and gdc_x for an older block implies gdc_and_quorum.

}
isolate ldc_properties5a = {

1. quorum_of_votes and gdc_x imply node_has_recent_lock.

}
isolate ldc_properties6a = { gdc_chain_condition is true for blocks and their parents such that the third component is a older block with gdc_x.

1. quorum_of_votes and gdc_x for some older block implies that the block has a lower round than the parent.
2. If gdc_chain_condition holds then the block is older than the parent.

}
isolate ldc_properties6b = {

1. quorum_of_votes and gdc imply gdc_chain_condition.

}
isolate ldc_properties6c = {

1. quorum_of_votes and gdc for an older block implies the block has lower round than the parent block.

}
isolate gdc_chain_lemma = { The ancestor relation is the reflexive transitive closure of the parent relation. The block_parent relation tries to define the parent relation in terms of the ancestor relation.

1. The round of a block is always greater than its parent.
2. The nil block has the lowest round.
3. ancestor is anti-symmetric.
4. ancestor is transitive.
5. If two blocks have a common ancestor, then one is the ancestor of the other.
6. Block's parent and block_parent have a correspondence.

}
isolate gdc_chain_lemma1 = { qov_after_gdc is about some new block getting a quorum of votes after a block is marked gdc.

1. Once a block has `qov_over_gdc`, it continues having that for larger rounds.

```
}
isolate gdc_chain_lemma2 = {
```

1. `block_has_qov` implies `quorum_of_votes`.

```
}
isolate gdc_chain_lemma2a = {
```

1. `block_has_qov` for a block and `gdc` for an older block implies that the older block has a lower round than the parent.

```
}
isolate gdc_chain_lemma3 = {
```

1. `block_has_qov` and `gdc` for an older block implies that the parent `block_has_qov`.

```
}
isolate gdc_chain_lemma4 = {
```

1. `block_has_qov` and `gdc` implies `block_has_qov` for the parent.
2. `block_has_qov`, `gdc`, and `qov_after_gdc` combined imply the ancestor relation.
3. Same as before, but the blocks are now non-`nil`.

```
}
isolate gdc_chain_lemma5 = {
```

1. Ancestor of parent is an ancestor.
2. `block_has_qov`, parent block with `gdc` and `qov_after_gdc` implies that the new block is a descendant of the `gdc` block.
3. `block_has_qov`, any old block with `gdc` and `qov_after_gdc` implies that the new block is a descendant of the `gdc` block.
4. `block_has_qov` and `gdc` for an older (non-genesis) block and `qov_after_gdc` being true for all rounds older than the new block's round implies that the new block is a descendant of the `gdc` block.
5. `block_has_qov` and `gdc` for an older block and `qov_after_gdc` being true for all rounds older than the new block's round implies that the new block is a descendant of the `gdc` block.

```
}
isolate gdc_chain_lemma6 = {
```

1. `block_has_qov` and `gdc` for an older (non-genesis) block and `qov_after_gdc` being true for all blocks older than the new block implies that the new block is a descendant of the `gdc` block.

2. Once a block has `qov_over_gdc`, it continues having that for the next round.

3. `qov_after_gdc` is true for round zero.

}
`isolate gdc_chain_lemma7 = { round_induction` defines an induction tactic that we will use to prove the property that follows. This needs to be done because induction is not available in first-order logic.

1. `qov_after_gdc` is true for all rounds.

2. `block_has_qov` and `gdc` for an older (non-genesis) block and `qov_after_gdc` being true for all rounds older than the new block's round implies that the new block is a descendant of the `gdc` block.

3. `qov_after_gdc` is true for all block rounds.

4. `gdc` being set, and a later block getting a `block_has_qov` implies the `gdc` block is an ancestor of the new block.

}
`isolate gdc_chain_lemma8 = {`

1. Tip of the blockchain has `gdc_x`.

2. `pred_of_chain_height` is one less than the `tip_of_chain_height`.

3. For nodes that have not locked any block, the uncommitted forest must be empty.

4. Chain height comparison.

}
`isolate gdc_chain_lemma9 = {`

1. The tip of the blockchain is `gdc`.

2. Tip of the blockchain has the correct predecessor.

3. The block at height zero in the blockchain is the genesis block.

4. The tip of chain `block_has_qov`.

5. For any two nodes, both have `gdc` and `block_has_qov` for their tip of the chain blocks.

6. For any two nodes with one tip of the chain having a round smaller than the other tip of the chain, the smaller tip of the chain block has `gdc`, the longer `block_has_qov`.

7. For any two nodes with one tip of the chain having a round smaller than the other tip of the chain, the smaller tip of the chain block having `gdc`, the longer having `block_has_qov`, the smaller tip is the ancestor of the longer.

8. Same thing as above.

}
isolate gdc_chain_lemma10 = {

1. If two oblocks have the same round number for the tip of the chain block, then they are the same block.

2. For any two good nodes, one tip of the chain is the ancestor of the other.

}
isolate ancestor_not_sibling = { The `orphan` relation is true for blocks that have no parent.

1. If a block is the ancestor of the other, but they have the same parent, then they are the same block.

}
isolate blockchain_ancestor = { The `height_induction` tactic is also an induction tactic, but this time we use `height_t` rather than `round_t` as before. The relation `blockchain_height_ancestor` is true if all older blocks in the blockchain are ancestors of the block at the given height.

1. The block at height zero is its own ancestor (genesis).

2. `blockchain_height_ancestor` is true for height zero.

3. For two successive heights in the blockchain, one block is the parent of the other.

4. Same as before, but with `block_parent` instead.

5. Induction step for `blockchain_height_ancestor`.

6. `blockchain_height_ancestor` inductive proof.

}
isolate blockchain_ancestor1 = {

1. `blockchain_height_ancestor` is true for all heights.

}
isolate blockchain_safety_lemma0 = {

1. If there are non-`nil` blocks in the chain, then the `chain_size` is non-zero.
2. If there is a non-`nil` block in the chain, then the tip of the chain is also non-`nil`.
3. Every block in the blockchain is an ancestor of the tip of the chain block.

}
`isolate blockchain_safety_lemma1 = {`

1. `chain_size` is one-more than the `tip_of_chain_height` (because the chain is zero-indexed).
2. Blocks beyond `chain_size` are all `nil`.
3. If there is a non-nill block, then `chain_size` is non-zero.
4. Across two different local blockchains, one tip of the chain is the ancestor of the other.
5. Consequence of `ancestor` being reflexive.
6. Basic result about `ancestor`.
7. Across different local blockchains and the same height, one block is the ancestor of the other.

}
`isolate blockchain_safety_lemma2 = {` `blockchain_safe` is true if the non-`nil` blocks at the same height are the same across blockchains.

1. `blockchain` at height zero is the correct (genesis) block.
2. The height zero blocks have some sort of ancestry among themselves.
3. If `blockchain_safe` is true for some height, then `blockchain` at height zero is the correct (genesis) block.
4. `blockchain_safe` is true for height zero.
5. Basic property about the successor function for heights.
6. Ancestry claims across arbitrary (but same) heights.
7. Same parent claim across arbitrary (but same) heights.
8. Same block claim across arbitrary (but same) heights.
9. Induction step for `blockchain_safe`.
10. Proof of `blockchain_safe` for all heights.

```
}
isolate global_properties = {
```

1. If a non-`nil` block exists in a blockchain, then that node has a QC corresponding to it for which `qc_processed` is true.
2. Any two blocks across any two nodes with the same round implies that the block is the same.
3. If a block was voted for, then its parent was locked.
4. Local blockchain across any two nodes are either the same or one is a subchain of the other.

```
}
}
```

3. Conclusion

In this thesis, we looked at the safety of SupraBFT. Unfortunately, through testing, it has been found that the protocol is not live. However, the protocol is in development and has since gone through multiple revisions. Revising the safety proof (and proving liveness) is an ongoing task.

Bibliography

[1] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[2] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, University of Guelph, 2016.

[3] Miguel Castro, Barbara Liskov, et al. A correctness proof for a practical byzantine-fault-tolerant replication algorithm. Technical report, Technical Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science, 1999.

[4] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[5] Alonzo Church. Correction to a note on the entscheidungsproblem. *The journal of symbolic logic*, 1(3):101–102, 1936.

[6] Alonzo Church. A note on the entscheidungsproblem. *The journal of symbolic logic*, 1(1):40–41, 1936.

[7] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[8] Yeting Ge and Leonardo De Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26-July 2, 2009. Proceedings 21*, pages 306–320. Springer, 2009.

[9] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *International Conference on Financial Cryptography and Data Security*, pages 296–315. Springer, 2022.

[10] Leander Jehl. Formal verification of hotstuff. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 197–204. Springer, 2021.

[11] Daniel Kroening and Ofer Strichman. *Decision procedures*. Springer, 2016.

[12] Harry R Lewis. Complexity results for classes of quantificational formulas. *Journal of Computer and System Sciences*, 21(3):317–353, 1980.

[13] Giuliano Losa. Formally verifying the tendermint blockchain protocol, Jul 2021. Retrieved 2022-12-05, https://web.archive.org/web/20221205110135/https://galois.com/blog/2021/07/formally-verifying-the-tendermint-blockchain-protocol/.

[14] Kenneth L McMillan. Decidability, May 2018. Retrieved 2023-05-23, https://web.archive.org/web/20230522085937/https://kenmcmil.github.io/ivy/decidability.html.

[15] Kenneth L McMillan and Oded Padon. Deductive verification in decidable fragments with ivy. In *Static Analysis: 25th International Symposium, SAS 2018, Freiburg, Germany, August 29–31, 2018, Proceedings 25*, pages 43–55. Springer, 2018.

[16] Kenneth L McMillan and Oded Padon. Ivy: a multi-modal verification tool for distributed algorithms. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II 32*, pages 190–202. Springer, 2020.

[17] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 614–630, 2016.

[18] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

A. The protocol

Algorithm 1: SupraBFT Round Transition

```
1  procedure advanceToRound(r, b_{r-1}, qc, tc_{r-2}) do
2      r_c := r
3      resetRoundTimer()
4
5      if b_{r-1} ≠ ⊥ then
6          /* Fast path.                                                    */
7          ν := Prepare(b_{r-1})
8          broadcast(ν)
9          voted := ν
10
11     if leader(r_c) = id ∧ (qc ≠ ⊥ ∨ tc_{r-2} ≠ ⊥) then
12         lb := if locked = ⊥ then ⊥ else locked.b
13         pb := if voted ≠ ⊥ ∧ b_{r-1} ≠ ⊥ ∧ voted.b = b_{r-1} then b_{r-1} else lb
14         pb := if tc_{r-2} ≠ ⊥ then getHighestQC(tc_{r-2}.c).b else pb
15         b := Block(r_c, digest(pb), getTxs())
16         p := Proposal(b, qc, tc_{r-2})
17         broadcast(p)
```

Algorithm 2: SupraBFT Locking and Committing

```
18 procedure tryLockAndCommit(qc := PrepareQC(b, c)) do
19     if b.r = locked.b.r + 1 ∧ b.parent = digest(locked.b) then
20         /* 2-chain commit. Commit all uncommitted ancestors of
               locked.                                              */
21         commit(locked)
22
23     if b.r > locked.b.r then
24         /* Newer PrepareQC for the same height or higher than locked.
               */
25         locked := qc
26
27 upon having qc := PrepareQC(b, c) either received in some protocol message or
     constructed from n − f Prepare(b) messages from unique senders
29     | validateQC(b.r, c) ∧ b.parent ≠ ⊥
30 do
31     store(qc)
32     waitForAncestorQCs(qc)
33     broadcast(qc)
34     tryLockAndCommit(qc)
```

Algorithm 3: SupraBFT Fast Path

```
35 upon having Proposal(b, qc, ⊥) from l_{r_c}
37     | b.r = r_c ∧ roundTimer < Δ ∧ digest(qc.b) = locked.b.parent ∧ locked.b.r
     = r_c − 1
39     ∧ b.parent = digest(locked.b)
40 do
41     advanceToRound(r_c + 1, b, locked, ⊥)
```

Algorithm 4: SupraBFT Fallback

```
42 procedure timeout(r) do
43     t := Timeout(r, locked)
44     broadcast(t)
45
46 /* Fallback.                                                    */
47 upon roundTimer = Δ ∧ locked.b.r ≠ r_c − 1
48 do
49     timeout(r_c − 1)
50
51 /* TimeoutQC processing.                                        */
52 upon having tc := TimeoutQC(r, c) either received in a Proposal from l_{r+1} or
     constructed from n − f unique Timeout(r, _) messages
54     | r > r_c − 2 ∧ validateTC(r, c)
55 do
56     advanceToRound(r + 2, ⊥, ⊥, tc)
57
58 /* Fallback recovery.                                          */
59 upon having Proposal(b, ⊥, tc) from l_r with qc' = getHighestQC(tc.c)
61     | b.r = r_c ∧ roundTimer < Δ ∧ tc.r = b.r − 2 ∧ b.parent = digest(qc'.b)
63     ∧ b.parent = digest(locked.b)
64 do
65     advanceToRound(r_c + 1, b, qc', ⊥)
```

Algorithm 5: SupraBFT Round Synchronisation

```
66 upon finishing PrepareQC Processing for qc := PrepareQC(b, c)
68     | b.r + 1 > r_c ∧ validateQC(b.r, c)
69 do
70     /* Received a QC for a higher round.                        */
71     if previously processed qc' := PrepareQC(b', c')
73         | b' = b.parent ∧ validateQC(b'.r, c') then
74         advanceToRound(b.r + 1, b, qc', ⊥)
75     else
76         advanceToRound(b.r + 1, ⊥, ⊥, ⊥)
77
78 upon roundTimer ≥ Δ ∧ locked.b.r = r_c − 1
79 do
80     /* Did not receive a valid Proposal for this round.        */
81     advanceToRound(b.r + 1, ⊥, ⊥, ⊥)
```

Algorithm 6: SupraBFT Genesis

```
/* Genesis.                                                              */
upon having Proposal(b, ⊥, ⊥) from l_r
    | b.r = r_c ∧ roundTimer < Δ ∧ b.parent = ⊥ ∧ locked = ⊥ ∧ voted = ⊥
do
    advanceToRound(r_c + 1, b, ⊥, tc)

/* Genesis fallback recovery.                                            */
upon having Proposal(b, ⊥, tc) from l_r
    | b.r = r_c ∧ roundTimer < Δ ∧ b.parent = ⊥ ∧ locked = ⊥ ∧ tc.r = b.r − 2
    ∧ validateTC(tc.r, tc.c)
do
    advanceToRound(r_c + 1, b, ⊥, tc)

/* Genesis + 1 fast path.                                                */
upon having Proposal(b, ⊥, ⊥) from l_r
    | b.r = r_c ∧ roundTimer < Δ ∧ b.parent = digest(locked.b)
do
    advanceToRound(r_c + 1, b, ⊥, ⊥)

upon having qc := PrepareQC(b, c) either received in some protocol message
  from v′ or constructed from n − f Prepare(b) messages from unique senders
    | validateQC(b.r, c) ∧ b.parent = ⊥
    ∧ (locked = ⊥ ∨ locked.b.parent = ⊥ ∧ b.r > locked.b.r)
do
    broadcast(qc)
    locked := qc
```

B. Ivy code

B.1. ubd_seq.ivy

```
#lang ivy1.8

include order

module ubd_seq = {
    type this
    alias t = this

    # returns the least key greater than x
    action next(x:t) returns (y:t)

    # returns the greatest key less than x
    action prev(x:t) returns (y:t)

    relation succ(X:t,Y:t)

    definition [successor_definition] succ(X,Z) = X < Z & forall Y:t. (X < Y ->
        (Z < Y | Z = Y))

    function max2(X:t,Y:t) = Y if X <= Y else X
    function mod(X:this,Y:this) = X - X / Y * Y

    object spec = {

        instantiate totally_ordered_with_zero(t)

        after next {
            assert x < y & (x < Y -> y <= Y);
            assert succ(x,y)
        }
        before prev {
            assert 0 < x
        }
        after prev {
            assert y < x & (Y < x -> Y <= y);
            assert succ(y,x)
        }
    }
```

```
    object impl = {

        interpret t -> nat

        implement next {
            y := x + 1
        }
        implement prev {
            y := x - 1
        }
    }

    isolate iso = impl,spec,successor_definition

    # When testing, use the integer implementation.

    attribute test = impl

}
```

B.2. domain_model.ivy

```
#lang ivy1.8

include ubd_seq

instance round_t : ubd_seq
instance height_t : ubd_seq

instance node_t : iterable
function leader(R:round_t) : node_t
relation is_good(N:node_t)

isolate hash_t = {
    type this

    individual nil: hash_t
}

isolate block_t = {
    type this = struct {
        round : round_t,
        parent : hash_t
    }
    individual nil : block_t
    function hash(B:block_t) : hash_t
    action consBlock(p:hash_t, r:round_t, n:node_t) returns (b:block_t)
```

```
    specification {
        axiom forall B:block_t. B = nil <-> B.parent = hash_t.nil

        axiom hash(X) = hash(Y) -> X = Y
        axiom hash(X) ~= hash_t.nil

        before consBlock(p:hash_t, r:round_t, n:node_t) returns (b:block_t) {
            require ~is_good(n) | p ~= hash_t.nil;
        }

        after consBlock(p:hash_t, r:round_t) returns (b:block_t) {
            ensure b ~= block_t.nil
        }
    }

    implementation {
        implement consBlock(p:hash_t, r:round_t) returns (b:block_t) {
            b.round := r;
            b.parent := p;
        }
    }
}

isolate quorum_t = {
    type this
    relation member(N:node_t, Q:quorum_t)

    specification {
        # only use in proofs
        axiom forall Q:quorum_t. exists N:node_t. is_good(N) & member(N, Q)

        axiom forall Q1:quorum_t. forall Q2:quorum_t. exists N:node_t. is_good(N
            ) & member(N, Q1) & member(N, Q2)
    }
}

isolate voted_t = {
    type this = struct {
        block : block_t
    }
    individual nil : voted_t

    specification {
        axiom forall V:voted_t. V = nil <-> V.block = block_t.nil
    }
}

isolate cert_t = {
    type this = struct {
```

```
        block : block_t,
        # every member of cert should have made rcvd_prepare() true for this
            round
        round : round_t,
        cert : quorum_t
    }
    individual nil : cert_t

    specification {
        axiom forall Qc:cert_t. Qc = nil <-> Qc.block = block_t.nil
        axiom forall Qc:cert_t. Qc ~= nil <-> Qc.round ~= nil.round
        axiom forall N:node_t. quorum_t.member(N, nil.cert)
    }
}

isolate timeout_t = {
    type this = struct {
        round : round_t,
        qc : cert_t
    }
    individual nil : timeout_t

    specification {
        axiom forall T:timeout_t. T ~= nil <-> T.round ~= nil.round
        axiom forall T:timeout_t. T = nil <-> T.qc = cert_t.nil
    }
}

isolate timeout_cert_t = {
    type this = struct {
        round : round_t,
        cert : quorum_t,
        highestQC : cert_t
    }
    individual nil : timeout_cert_t

    specification {
        axiom forall Tc:timeout_cert_t. Tc ~= nil <-> Tc.round ~= nil.round
        axiom forall N:node_t. quorum_t.member(N, nil.cert)
        axiom forall Tc:timeout_cert_t. Tc = nil <-> Tc.highestQC = cert_t.nil
    }
}

isolate proposal_t = {
    type this = struct {
        block : block_t,
        qc : cert_t,
        tc : timeout_cert_t
    }
```

```
    individual nil : proposal_t

    specification {
        axiom forall P:proposal_t. P = nil <-> P.block = block_t.nil
        axiom nil.qc = cert_t.nil
        axiom nil.tc = timeout_cert_t.nil
    }
}
```

B.3. network_model.ivy

```
#lang ivy1.8

include udp

include domain_model

object msg_kind = {
    type this = {proposal, prepare, qc, timeout, tc}
}

object msg = {
    type this = struct {
        kind : msg_kind,
        prop : proposal_t,
        prep : block_t,
        qc : cert_t,
        t : timeout_t,
        tc : timeout_cert_t,
        src : node_t
    }
}

instance net : udp_simple(node_t, msg)

isolate shim = {
    # In order not repeat the same code for each handler, we use a handler
    # module parameterized by the type of message it will handle. Below we
    # instantiate this module for the five types of messages of SupraBFT
    module handler(p_kind) = {
        action handle(dst:node_t, m:msg)
        object spec = {
            before handle {
                assert sent(m, dst) & m.kind = p_kind
            }
        }
    }
```

```
instance proposal_handler : handler(msg_kind.proposal)
instance prepare_handler : handler(msg_kind.prepare)
instance qc_handler : handler(msg_kind.qc)
instance timeout_handler : handler(msg_kind.timeout)
instance tc_handler : handler(msg_kind.tc)

relation sent(M:msg, N:node_t)

action broadcast(src:node_t, m:msg)
action send(src:node_t,dst:node_t, m:msg)

specification {
    after init {
        sent(M, D) := false;
    }
    before broadcast {
        sent(m, D) := true
    }
    before send {
        sent(m, dst) := true
    }
}

# Here we give an implementation of it that satisfies its specification:
implementation {
    implement net.recv(dst:node_t ,m:msg) {
        if m.kind = msg_kind.proposal {
            call proposal_handler.handle(dst, m)
        } else if m.kind = msg_kind.prepare {
            call prepare_handler.handle(dst, m)
        } else if m.kind = msg_kind.qc {
            call qc_handler.handle(dst, m)
        } else if m.kind = msg_kind.timeout {
            call timeout_handler.handle(dst, m)
        } else if m.kind = msg_kind.tc {
            call tc_handler.handle(dst, m)
        }
    }

    # broadcast sends to all nodes, including the sender.
    implement broadcast {
        var iter := node_t.iter.create(0);
        while ~iter.is_end
        invariant net.spec.sent(M,D) -> sent(M,D) {
            var n := iter.val;
            call net.send(src, n, m);
            iter := iter.next;
        }
    }
```

```
        implement send {
            call net.send(src, dst, m)
        }

        private {
            invariant net.spec.sent(M, D) -> sent(M, D)
        }
    }
# to prove that the shim implementation satisfies the shim specification, we
    rely on the specification of net and node.
} with net.spec, node_t
```

B.4. algorithm.ivy

```
#lang ivy1.8

include domain_model
include network_model

module supraBFT(global_view) = {
object validator(id:node_t) = {
    individual locked : cert_t
    individual voted : voted_t
    individual r_c : round_t

    individual timed_out : bool

    function blockchain (H:height_t) : block_t
    individual chain_size : height_t
    function blockchain_temp (H:height_t) : block_t

    relation rcvd_proposal(P:proposal_t, Src:node_t)
    relation rcvd_prepare(B:block_t, Src:node_t)
    relation rcvd_qc(Qc:cert_t)
    relation rcvd_timeout(T:timeout_t, Src:node_t)
    relation rcvd_tc(Tc:timeout_cert_t)

    function valid_qc(Qc:cert_t) : bool
    # this function checks that all members in Qc have voted for the block
        contained in Qc. It is done here by directly checking the node_has_voted
         relation. In actual implementation, this should be done by examining
        the contents of Qc.
    definition valid_qc(Q:cert_t) = forall N:node_t. quorum_t.member(N,Q.cert)
        -> global_view.node_has_voted(N, Q.block)

    function valid_tc(Tc:timeout_cert_t) : bool
```

```
definition valid_tc(Tc:timeout_cert_t) = (forall N:node_t. quorum_t.member(N
    , Tc.cert) -> exists T:timeout_t. T.round = Tc.round & global_view.
    node_sent_timeout(N, T) & Tc.highestQC.round >= T.qc.round) & (exists N:
    node_t. exists T:timeout_t. quorum_t.member(N, Tc.cert) & T.round = Tc.
    round & global_view.node_sent_timeout(N, T) & Tc.highestQC = T.qc) &
    valid_qc(Tc.highestQC)

relation forest_store(Qc:cert_t,H:height_t)
relation forest_ancestor(Qd:cert_t,Hd:height_t,Qa:cert_t,Ha:height_t) #Qa is
     an ancestor of Qd
relation forest_strictAncestor(Qd:cert_t,Hd:height_t,Qa:cert_t,Ha:height_t)
relation forest_root(Q:cert_t,H:height_t)
relation forest_leaf(Q:cert_t,H:height_t)
relation forest_succ(Qd:cert_t,Hd:height_t,Qa:cert_t,Ha:height_t)
relation qc_processed(Q:cert_t)

definition forest_strictAncestor(Qd, Hd, Qa, Ha) = forest_store(Qd, Hd) &
    forest_store(Qa, Ha) & forest_ancestor(Qd, Hd, Qa, Ha) & (Qd ~= Qa | Hd
    ~= Ha)

definition forest_root(Q, H) = forest_store(Q, H) & forall Qa:cert_t. forall
     Ha:height_t. forest_ancestor(Q, H, Qa, Ha) -> (Q = Qa & H = Ha)

definition forest_leaf(Q, H) = forest_store(Q, H) & forall Qd:cert_t. forall
     Hd:height_t. forest_ancestor(Qd, Hd, Q, H) -> (Qd = Q & Hd = H)

definition forest_succ(Qd, Hd, Qa, Ha) = forest_ancestor(Qd, Hd, Qa, Ha) & (
    Qd ~= Qa | Hd ~= Ha) & forall Qaa:cert_t. forall Haa:height_t. ((
    forest_ancestor(Qd, Hd, Qaa, Haa) & (Qd ~= Qaa | Hd ~= Haa)) ->
    forest_ancestor(Qa, Ha, Qaa, Haa))

relation forest_heights(Qf:cert_t, Hf:height_t, Ha:height_t)

definition forest_heights(Qf, Hf, Ha) = exists Qa:cert_t. forest_ancestor(Qf
    ,Hf,Qa,Ha)

# These relations are used to store Qcs that are ancestors of currently
    locked QC, to implement 2-chain commit
relation commit_candidates(Q:cert_t, H:height_t)

# this is used to store the chain_size at which 2-chain commit is executed.
    It is used to write supporting invariants
individual chain_size_2cc : height_t
individual dc_basis_qc : cert_t
individual dc_basis_block : block_t
individual tip_of_chain_qc : cert_t
individual tip_of_chain_height : height_t
individual qp_when_adding_qc : cert_t
individual pred_of_chain_height : height_t
```

```
individual locked_height : height_t

after init {
    locked := cert_t.nil;
    locked.block := block_t.nil;
    voted := voted_t.nil;
    voted.block := block_t.nil;
    r_c := 0;
    locked_height := 0;

    timed_out := false;

    blockchain(H) := block_t.nil;
    chain_size := 0;

    rcvd_proposal(P, Src) := false;
    rcvd_prepare(B, Src) := false;
    rcvd_qc(Qc) := false;
    rcvd_timeout(T, Src) := false;
    rcvd_tc(Tc) := false;

    forest_store(Q, H) := false;
    forest_ancestor(Q1, H1, Q2, H2) := false;

    qc_processed(Q) := false;

    commit_candidates(Q,H) := false;
    chain_size_2cc := 0;
    dc_basis_qc := cert_t.nil;
    dc_basis_block := block_t.nil;
    tip_of_chain_qc := cert_t.nil;
    tip_of_chain_height := 0;
    qp_when_adding_qc := cert_t.nil;
    pred_of_chain_height := 0;

    assume forall B:block_t. B.parent = block_t.hash(block_t.nil) if B ~=
        block_t.nil else hash_t.nil;

    assume forall B:block_t. B.round = round_t.next(0) if B ~= block_t.nil
        else 0;
}

implement shim.proposal_handler.handle(m:msg) {
    if m.prop ~= proposal_t.nil {
        if (m.prop.qc ~= cert_t.nil -> valid_qc(m.prop.qc)) & (m.prop.tc ~=
            timeout_cert_t.nil -> valid_tc(m.prop.tc)) {
            rcvd_proposal(m.prop, m.src) := true;
            if m.prop.qc ~= cert_t.nil {
```

```
            rcvd_qc(m.prop.qc) := true;
        }
        if m.prop.tc ~= timeout_cert_t.nil {
            rcvd_tc(m.prop.tc) := true;
        }
    }
}
}
implement shim.prepare_handler.handle(m:msg) {
    if m.prep ~= block_t.nil {
        rcvd_prepare(m.prep, m.src) := true;
    }
}
implement shim.qc_handler.handle(m:msg) {
    if m.qc ~= cert_t.nil {
        if valid_qc(m.qc) {
            rcvd_qc(m.qc) := true;
        }
    }
}
implement shim.timeout_handler.handle(m:msg) {
    # here we check global view to ensure that the sender of this message
        has really sent this timeout message. In implementation, this is
        done by checking the cryptographic authenticity of the message
    if m.t ~= timeout_t.nil {
        if m.t.qc ~= cert_t.nil & valid_qc(m.t.qc) {
            rcvd_timeout(m.t, m.src) := true;
        }
    }
}
implement shim.tc_handler.handle(m:msg) {
    if m.tc ~= timeout_cert_t.nil {
        if valid_tc(m.tc) {
            rcvd_tc(m.tc) := true;
        }
    }
}

action broadcast_proposal(p:proposal_t) = {
    var m : msg;
    m.kind := msg_kind.proposal;
    m.prop := p;
    m.src := id;

    call shim.broadcast(id, m);
}

action broadcast_prepare(b:block_t) = {
    var m : msg;
```

```
    m.kind := msg_kind.prepare;
    m.prep := b;
    m.src := id;

    call shim.broadcast(id, m);
}

action broadcast_qc(qc:cert_t) = {
    var m : msg;
    m.kind := msg_kind.qc;
    m.qc := qc;
    m.src := id;

    call shim.broadcast(id, m);
}

action broadcast_timeout(t:timeout_t) = {
    var m : msg;
    m.kind := msg_kind.timeout;
    m.t := t;
    m.src := id;

    call shim.broadcast(id, m);
}

action broadcast_tc(tc:timeout_cert_t) = {
    var m : msg;
    m.kind := msg_kind.tc;
    m.tc := tc;
    m.src := id;

    call shim.broadcast(id, m);
}

export action aggregate_prepares(qc:cert_t) = {
    require forall N:node_t. quorum_t.member(N, qc.cert) -> rcvd_prepare(qc.
        block, N);

    rcvd_qc(qc) := true;
}

export action aggregate_timeout(tc:timeout_cert_t) = {
    require forall N:node_t. quorum_t.member(N, tc.cert) -> exists T:
        timeout_t. T.round = tc.round & rcvd_timeout(T, N) & tc.highestQC.
        round >= T.qc.round;
    require exists N:node_t. exists T:timeout_t. quorum_t.member(N, tc.cert)
         & T.round = tc.round & rcvd_timeout(T, N) & tc.highestQC = T.qc;
    require valid_qc(tc.highestQC);
```

```
    rcvd_tc(tc) := true;
}

# We simulate timer expiry by having the environment call this action, in
    which we set the boolean timed_out to true
export action timer_expiry = {
    require ~timed_out;

    timed_out := true;
}

export action byzantine_send = {
    require ~is_good(id);

    var m : msg;
    var dst : node_t;
    if ~is_good(m.src) {
        if m.kind = msg_kind.proposal & (m.prop.qc ~= cert_t.nil -> valid_qc
            (m.prop.qc)) & (m.prop.tc ~= timeout_cert_t.nil -> valid_tc(m.
            prop.tc)) {
            call shim.send(id, dst, m);
        }
        if m.kind = msg_kind.prepare {
            var r : round_t;
            var bl: block_t;
            call shim.send(id, dst, m);
            call global_view.node_voted(m.src,m.prep,r,bl);
        }
        if m.kind = msg_kind.qc & valid_qc(m.qc) {
            var ok : bool;
            ok := true;
            ok := ok & forall H:height_t. ~forest_store(m.qc,H);
            if ok {
                call shim.send(id, dst, m);
            }
        }
        if m.kind = msg_kind.timeout & valid_qc(m.t.qc) {
            call shim.send(id, dst, m);
            call global_view.node_timeout(m.src,m.t);
        }
        if m.kind = msg_kind.tc & valid_tc(m.tc) {
            call shim.send(id, dst, m);
        }
    }
}

# Normal Path
action advanceToRound(r:round_t, b_r_1:block_t, qc:cert_t, tc_r_2:
    timeout_cert_t) = {
```

```
    require ~is_good(id) | qc = cert_t.nil | rcvd_qc(qc);
    require tc_r_2 = timeout_cert_t.nil | rcvd_tc(tc_r_2);
    require b_r_1 ~= block_t.nil -> b_r_1.parent = block_t.hash(locked.block
        );
    require forall B:block_t. forall R:round_t. (is_good(id) & global_view.
        node_has_locked_detail(id,B,R)) -> (B.round < locked.block.round | B
         = locked.block);

    r_c := r;
    timed_out := false;

    if (b_r_1 ~= block_t.nil) {
        call broadcast_prepare(b_r_1);
            call global_view.node_voted(id, b_r_1, r_c, locked.block);

        voted.block := b_r_1;
    }

    if id = leader(r_c) {
        var pb : block_t;
        pb := b_r_1 if b_r_1 ~= block_t.nil & voted.block = b_r_1 else
            locked.block;
        if tc_r_2 ~= timeout_cert_t.nil {
            pb := tc_r_2.highestQC.block;
        }
        var b : block_t;
        assert ~is_good(id) | block_t.hash(pb) ~= hash_t.nil;
        assume b.parent = block_t.hash(pb);
        b.round := r_c;
        var p : proposal_t;
        p.block := b;
        p.qc := qc;
        p.tc := tc_r_2;
        call broadcast_proposal(p);
    }
}

action tryLockAndCommit(qc:cert_t, hc:height_t, qp:cert_t) = {
    require qc ~= cert_t.nil & rcvd_qc(qc);
    require qc.block.parent ~= block_t.hash(block_t.nil);
    require forest_store(qc,hc);

    if round_t.succ(locked.block.round, qc.block.round) & qc.block.parent =
        block_t.hash(locked.block) {
        #2-chain commit. Commit all uncommitted ancestors of locked, which
            is the parent of qc
        blockchain_temp(H) := blockchain(H);
        commit_candidates(Q,H) := chain_size <= H & H < hc & forest_ancestor
            (qc,hc,Q,H);
```

```
        commit_candidates(qc,hc) := true;

        chain_size_2cc := chain_size;
        dc_basis_qc := qc;
        dc_basis_block := qc.block;
        tip_of_chain_qc := locked;
        tip_of_chain_height := locked_height;
        qp_when_adding_qc := qp;
        if (0 < tip_of_chain_height)
        {
            pred_of_chain_height := height_t.prev(tip_of_chain_height);
        }

        blockchain(H) := *;
        assume forall H:height_t. H < chain_size -> blockchain(H) =
            blockchain_temp(H);
        assume forall H:height_t. ( chain_size <= H & H < hc) -> exists Q:
            cert_t. commit_candidates(Q,H) & Q.block = blockchain(H);
        assume forall H:height_t. hc <= H -> blockchain(H) = block_t.nil;

        chain_size := hc;
        forest_store(Q,H) := Q=qc & H = hc;
        forest_ancestor(Q1,H1,Q2,H2) := Q1 = qc & H1 = hc & Q2 = qc & H2 =
            hc;
    }
    if locked.block.round < qc.block.round {
        locked := qc;
        locked_height := hc;
    }
}

export action process_prepareQC(qc:cert_t) = {
    require rcvd_qc(qc);

    var ok := true;
    ok := ok & qc.block ~= block_t.nil;
    ok := ok & qc.block.parent ~= block_t.hash(block_t.nil);
    ok := ok & forall H:height_t. blockchain(H) ~= qc.block;
    ok := ok & forall Q:cert_t. forall H:height_t. forest_store(Q,H) -> Q.
        block ~= qc.block;
    ok := ok & forall Q:cert_t. qc_processed(Q) -> Q.block ~= qc.block;
    if ok {
        if some h:height_t. height_t.succ(h, chain_size) & qc.block.parent =
             block_t.hash(blockchain(h)) {
            # this qc is a candidate for the child of the last committed
                block on the chain. start a new tree in the forest
            forest_store(qc, chain_size) := true;
            forest_ancestor(qc, chain_size, qc, chain_size) := true;
            call broadcast_qc(qc);
```

```
            call tryLockAndCommit(qc,chain_size,cert_t.nil);
            call finished_prepareQC_processing(qc);
            if locked = qc {
                assert ~is_good(id) | qc.block.round < r_c;
                call global_view.node_locked_quorum(id, qc.block, qc.cert,
                    r_c);
            }
            qc_processed(qc) := true;
            call global_view.node_processed_qc(id,qc);
        } else {
            if some qp:cert_t. exists Hp:height_t. forest_store(qp, Hp) & qc.
                block.parent = block_t.hash(qp.block){
                # this qc is a candidate for the child of qp, which is in the
                     forest at height Hp
                if some hp:height_t. forest_store(qp, hp) & qc.block.parent =
                     block_t.hash(qp.block) {
                    forest_store(qc, height_t.next(hp)) := true;
                    forest_ancestor(qc, height_t.next(hp), qc, height_t.next(
                        hp)) := true;
                    forest_ancestor(qc, height_t.next(hp), qp, hp) := true;
                    forest_ancestor(qc, height_t.next(hp), Q, H) :=
                        forest_ancestor(qp, hp, Q, H) | (Q=qc & H = height_t.
                        next(hp));

                    call broadcast_qc(qc);
                    call tryLockAndCommit(qc,height_t.next(hp),qp);
                    call finished_prepareQC_processing(qc);
                    if locked = qc {
                        assert ~is_good(id) | qc.block.round < r_c;
                        call global_view.node_locked_quorum(id, qc.block, qc.
                            cert, r_c);
                    }
                    qc_processed(qc) := true;
                    call global_view.node_processed_qc(id,qc);
                }
            } else {
                #wait for ancestor QCs
            }
        }
    }
}

export action process_proposal(p:proposal_t) = {
    require rcvd_proposal(p, leader(r_c));
    require p.qc ~= cert_t.nil;
    require p.tc = timeout_cert_t.nil;

    var ok := true;
    ok := ok & p.block.round = r_c;
```

```
    ok := ok & ~timed_out;
    ok := ok & block_t.hash(p.qc.block) = locked.block.parent;
    ok := ok & round_t.succ(locked.block.round, r_c);
    ok := ok & p.block.parent = block_t.hash(locked.block);
    if ok {
        call advanceToRound(round_t.next(r_c), p.block, locked,
            timeout_cert_t.nil);
    }
}

# Fallback Path
action timeout(r:round_t) = {
    var t : timeout_t;
    t.round := r;
    t.qc := locked;
    call broadcast_timeout(t);
    call global_view.node_timeout(id, t);
}

export action fallback = {
    require r_c > 0;

    var ok := true;
    ok := ok & timed_out;
    ok := ok & ~round_t.succ(locked.block.round, r_c);
    if ok {
        call timeout(round_t.prev(r_c));
    }
}

export action timeout_synchronisation(t:timeout_t) = {
    # f + 1 unique timeout messages -> at least 1 honest timeout message
    require exists N:node_t. is_good(N) & rcvd_timeout(t, N);
    # this is bad, but do we really want another data structure storing
        every timeout message we sent
    require forall T:timeout_t. T.round = t.round -> ~global_view.
        node_sent_timeout(id, T);
    require ~timed_out;

    call timer_expiry;
    call timeout(t.round);
}

export action process_timeoutQC(tc:timeout_cert_t) = {
    require rcvd_tc(tc);
    require r_c > 0;
    require round_t.prev(r_c) > 0;

    var ok := true;
```

```
    ok := ok & tc.round > round_t.prev(round_t.prev(r_c));
    if ok {
        call advanceToRound(round_t.next(round_t.next(tc.round)), block_t.
            nil, cert_t.nil, tc);
    }
}

export action fallback_recovery(p:proposal_t) = {
    require rcvd_proposal(p, leader(r_c));
    require p.qc = cert_t.nil;
    require p.tc ~= timeout_cert_t.nil;
    require 0 < p.block.round;
    require 0 < round_t.prev(p.block.round);

    var ok := true;
    ok := ok & p.block.round = r_c;
    ok := ok & ~timed_out;
    ok := ok & p.tc.round = round_t.prev(round_t.prev(p.block.round));
    ok := ok & valid_qc(p.tc.highestQC);
    ok := ok & p.block.parent = block_t.hash(p.tc.highestQC.block);
    ok := ok & p.block.parent = block_t.hash(locked.block);
    if ok {
        rcvd_qc(p.tc.highestQC) := true;
        call advanceToRound(round_t.next(r_c), p.block, p.tc.highestQC,
            timeout_cert_t.nil)
    }
}

action finished_prepareQC_processing(qc:cert_t) = {
    require rcvd_qc(qc);
    require ~is_good(id) | qc.block.parent = block_t.hash(block_t.nil) |
        exists Qc2:cert_t. qc_processed(Qc2) & block_t.hash(Qc2.block) = qc.
        block.parent;

    var ok := true;
    ok := ok & round_t.next(qc.block.round) > r_c;
    if ok {
        voted.block := qc.block;
        if some qc2:cert_t. qc_processed(qc2) & block_t.hash(qc2.block) = qc
            .block.parent {
            call advanceToRound(round_t.next(qc.block.round), block_t.nil,
                qc2, timeout_cert_t.nil);
        }
        else {
            if qc.block.parent = block_t.hash(block_t.nil)
            {
                 call advanceToRound(round_t.next(qc.block.round), block_t.
                     nil, cert_t.nil, timeout_cert_t.nil);
            }
```

```
        }
    }
}

export action no_proposal = {
    require exists Qc:cert_t. rcvd_qc(Qc) & locked.block.parent = block_t.
        hash(Qc.block);

    var ok := true;
    ok := ok & timed_out;
    ok := ok & round_t.succ(locked.block.round, r_c);
    if ok {
        if some qc:cert_t. rcvd_qc(qc) & locked.block.parent = block_t.hash(
            qc.block) {
            call advanceToRound(round_t.next(r_c), block_t.nil, qc,
                timeout_cert_t.nil);
        }
    }
}

export action leader_failed(p:proposal_t) = {
    require rcvd_proposal(p, leader(r_c));
    require p.qc ~= cert_t.nil;
    require p.tc = timeout_cert_t.nil;
    require r_c > 0;

    var ok := true;
    ok := ok & p.block.round = r_c;
    ok := ok & voted.block.round < round_t.prev(r_c);
    ok := ok & block_t.hash(p.qc.block) = locked.block.parent;
    ok := ok & p.block.parent = block_t.hash(locked.block);
    if ok {
        call advanceToRound(round_t.next(r_c), p.block, locked,
            timeout_cert_t.nil);
    }
}

# Genesis
export action proposal_b0_correct(p:proposal_t) = {
    require rcvd_proposal(p, leader(r_c));
    require p.qc = cert_t.nil;
    require p.tc = timeout_cert_t.nil;

    var ok := true;
    ok := ok & p.block.round = r_c;
    ok := ok & ~timed_out;
    ok := ok & p.block.parent = block_t.hash(block_t.nil);
    ok := ok & locked = cert_t.nil;
    ok := ok & voted = voted_t.nil;
```

```
    if ok {
        call advanceToRound(round_t.next(r_c), p.block, cert_t.nil,
            timeout_cert_t.nil);
    }
}

export action proposal_b0_late(p:proposal_t) = {
    require rcvd_proposal(p, leader(r_c));
    require p.qc = cert_t.nil;
    require p.tc ~= timeout_cert_t.nil;
        require p.block.round > 0;
        require round_t.prev(p.block.round) > 0;

    var ok := true;
    ok := ok & p.block.round = r_c;
    ok := ok & ~timed_out;
    ok := ok & p.block.parent = block_t.hash(block_t.nil);
    ok := ok & locked = cert_t.nil;
    ok := ok & p.tc.round = round_t.prev(round_t.prev(p.block.round));
    if ok {
        call advanceToRound(round_t.next(r_c), p.block, cert_t.nil, p.tc);
    }
}

export action proposal_b1_correct(p:proposal_t) = {
    require rcvd_proposal(p, leader(r_c));
    require p.qc = cert_t.nil;
    require p.tc = timeout_cert_t.nil;

    var ok := true;
    ok := ok & p.block.round = r_c;
    ok := ok & ~timed_out;
    ok := ok & p.block.parent = block_t.hash(locked.block);
    if ok {
        call advanceToRound(round_t.next(r_c), p.block, cert_t.nil,
            timeout_cert_t.nil);
    }
}

export action b0_qc(qc:cert_t) = {
    require rcvd_qc(qc);

    var ok := true;
    ok := ok & qc.block.parent = block_t.hash(block_t.nil);
    ok := ok & qc.block ~= block_t.nil;
    ok := ok & (locked = cert_t.nil | (locked.block.parent = block_t.hash(
        block_t.nil) & qc.block.round > locked.block.round));
    ok := ok & forall Q:cert_t. qc_processed(Q) -> Q.block ~= qc.block;
    if ok {
```

```
        call broadcast_qc(qc);
        locked := qc;
        locked_height := 0;
        forest_store(qc,0) := true;
        forest_ancestor(qc,0,qc,0) := true;
        call finished_prepareQC_processing(qc);
        assert qc.block.round < r_c;
        call global_view.node_locked_quorum(id,qc.block,qc.cert, r_c);
        qc_processed(qc) := true;
        call global_view.node_processed_qc(id,qc);
    }
}

invariant forall B:block_t. forall N:node_t. rcvd_prepare(B, N) ->
    global_view.node_has_voted(N, B)

invariant forall T:timeout_t. forall N:node_t. rcvd_timeout(T, N) ->
    global_view.node_sent_timeout(N, T)

    invariant forall B:block_t. (global_view.node_has_voted(id, B) & is_good
        (id)) -> r_c > B.round

invariant forall Qc:cert_t. rcvd_qc(Qc) -> valid_qc(Qc)

invariant forall Q:cert_t. qc_processed(Q) -> rcvd_qc(Q)

invariant forall Tc:timeout_cert_t. rcvd_tc(Tc) -> valid_tc(Tc)

invariant forall P:proposal_t. forall N:node_t. rcvd_proposal(P,N) -> ((P.tc
     ~= timeout_cert_t.nil -> rcvd_tc(P.tc)) & (P.qc ~= cert_t.nil ->
    rcvd_qc(P.qc)))

invariant forall N:node_t. ~rcvd_proposal(proposal_t.nil, N)

invariant forall N:node_t. ~rcvd_prepare(block_t.nil, N)

invariant forall N:node_t. ~rcvd_timeout(timeout_t.nil, N)

invariant ~rcvd_qc(cert_t.nil)

invariant ~rcvd_tc(timeout_cert_t.nil)

invariant (is_good(id) & locked.block ~= block_t.nil) -> locked.block.round
    < r_c

    invariant (locked.block ~= block_t.nil & is_good(id)) -> (exists Rl:
        round_t. Rl <= r_c & global_view.node_has_locked_detail(id, locked.
        block,Rl))
```

```
    invariant (locked.block ~= block_t.nil & is_good(id)) -> (global_view.
        node_has_locked(id, locked.block))

invariant ((locked = cert_t.nil | locked.block.parent = block_t.hash(block_t
    .nil)) & is_good(id)) -> chain_size=0

invariant forall H:height_t. (chain_size <= H & is_good(id)) -> (blockchain(
    H) = block_t.nil)

    invariant forall H:height_t. (H < chain_size & is_good(id)) -> (
        blockchain(H) ~= block_t.nil)

invariant forall H:height_t. (is_good(id) & blockchain(H) ~= block_t.nil) ->
     exists Q:cert_t. (qc_processed(Q) & Q.block = blockchain(H))

invariant (locked ~= cert_t.nil & is_good(id)) -> forest_store(locked,
    locked_height)

invariant forall Q:cert_t. forall H:height_t. (forest_store(Q,H) & is_good(
    id)) -> (rcvd_qc(Q) & forest_ancestor(Q,H,Q,H) & Q.block ~= block_t.nil
    & qc_processed(Q))

invariant forall Qd:cert_t. forall Qa:cert_t. forall Hd:height_t. forall Ha:
    height_t. (forest_ancestor(Qd,Hd,Qa,Ha) & is_good(id)) -> (forest_store(
    Qd,Hd) & forest_store(Qa,Ha))

invariant forall H:height_t. forall Hf:height_t. forall Qf:cert_t. (
    forest_store(Qf,Hf) & is_good(id)) -> Qf.block ~= blockchain(H)

invariant forall Hf:height_t. forall Qf:cert_t. (chain_size = 0 &
    forest_root(Qf,Hf) & is_good(id)) -> (Hf = 0 & Qf.block.parent = block_t
    .hash(block_t.nil))

invariant forall Hf:height_t. forall Qf:cert_t. (Hf = 0 & forest_root(Qf,Hf)
     & is_good(id)) -> (chain_size = 0 & Qf.block.parent = block_t.hash(
    block_t.nil))

invariant forall Q:cert_t. forall H:height_t. (forest_store(Q,H) & is_good(
    id)) -> chain_size <= H

invariant forall Q:cert_t. (is_good(id) & forest_store(Q,chain_size) ->
    forest_root(Q,chain_size))

invariant forall Hf:height_t. forall Qf:cert_t. forall Hc:height_t. (
    forest_root(Qf,Hf) & is_good(id) & height_t.succ(Hc,chain_size)) -> (Hf
    = chain_size & Qf.block.parent = block_t.hash(blockchain(Hc)))

invariant forall Qd:cert_t. forall Qa:cert_t. forall Hd:height_t. forall Ha:
    height_t. (forest_ancestor(Qd,Hd,Qa,Ha) & height_t.succ(Ha,Hd) & is_good
```

```
        (id)) -> Qd.block.parent = block_t.hash(Qa.block)

    invariant forall Qd:cert_t. forall Qa:cert_t. forall Hd:height_t. forall Ha:
        height_t. (forest_store(Qd,Hd) & forest_store(Qa,Ha) & is_good(id)) -> (
        forest_succ(Qd,Hd,Qa,Ha) -> (Qd.block.parent = block_t.hash(Qa.block) &
        height_t.succ(Ha,Hd)))

    invariant [blockchain_parent] forall H1:height_t. forall H2:height_t. (
        height_t.succ(H1, H2) & H2 < chain_size & is_good(id)) -> blockchain(H2)
        .parent = block_t.hash(blockchain(H1))

    invariant forall Q1,Q2:cert_t. forall H1,H2:height_t. (forest_ancestor(Q1,H1
        ,Q2,H2) & is_good(id)) -> forest_store(Q1,H1) & forest_store(Q2,H2)

    invariant forall Q1:cert_t. forall Q2:cert_t. forall H1:height_t. forall H2:
        height_t. (forest_store(Q1,H1) & forest_store(Q2,H2) & Q1.block = Q2.
        block & is_good(id)) -> (Q1 = Q2 & H1=H2)

    invariant forall Qd,Qa:cert_t. forall Hd,Ha:height_t. (forest_strictAncestor
        (Qd,Hd,Qa,Ha) & is_good(id)) -> Ha < Hd

    invariant forall Q1:cert_t. forall Q2:cert_t. forall H1:height_t. forall H2:
        height_t. (forest_ancestor(Q1,H1,Q2,H2) & H1 = H2 & is_good(id)) -> (Q1
        = Q2)

    invariant forall Q1,Q2:cert_t. forall H1,H2:height_t. is_good(id) ->((
        forest_ancestor(Q1,H1,Q2,H2) & forest_ancestor(Q2,H2,Q1,H1)) <-> (Q1 =
        Q2 & H1 = H2 & forest_store(Q1,H1) & forest_store(Q2,H2)))

    invariant forall Q1,Q2,Q3: cert_t. forall H1,H2,H3:height_t. (
        forest_ancestor(Q1,H1,Q2,H2) & forest_ancestor(Q2,H2,Q3,H3) & is_good(id
        )) -> forest_ancestor(Q1,H1,Q3,H3)

    invariant forall Q1,Q2,Q3: cert_t. forall H1,H2,H3:height_t. (
        forest_ancestor(Q1,H1,Q2,H2) & forest_ancestor(Q1,H1,Q3,H3) & is_good(id
        )) -> (forest_ancestor(Q2,H2,Q3,H3) | forest_ancestor(Q3,H3,Q2,H2))
} # object validator
} # module supraBFT
```

B.5. global_view.ivy

```
#lang ivy1.8

include domain_model

module global_view = {
        relation node_has_voted_detail(N:node_t, B:block_t, R:round_t, Bl:
            block_t) #node N has voted for block B in round R, when it had
```

```
    locked the block Bl

relation node_has_voted(N:node_t, B:block_t)

relation node_has_locked_detail(N:node_t, B:block_t, R:round_t)
relation node_has_locked(N:node_t, B:block_t) # node N has received
    quorum for block B

relation node_sent_timeout(N:node_t, T:timeout_t) # node N has sent a
    timeout message T

relation node_has_locked_recently(N:node_t, B:block_t, R:round_t)

definition node_has_locked_recently(N:node_t, B:block_t, R:round_t) =
    exists Rl:round_t. forall Rs:round_t. forall Br:block_t. (Rl < R &
    gv.node_has_locked_detail(N,B,Rl) & ((Rl < Rs & Rs < R) -> ~gv.
    node_has_locked_detail(N,Br,Rs)) & ((gv.node_has_locked_detail(N,Br,
    Rl) & Br ~= B) -> Br.round < B.round))

relation node_has_processed_qc(N:node_t, Qc:cert_t)

relation quorum_of_votes(Bc:block_t, Bp:block_t)
relation quorum_of_recent_locks(Bp:block_t, R:round_t)

action node_voted(n:node_t, b:block_t, r:round_t, bl:block_t)
action node_locked_quorum(n:node_t, b:block_t, q:quorum_t, r:round_t)
action node_timeout(n:node_t, t:timeout_t)
action node_processed_qc(n:node_t, qc:cert_t)

after init {
       node_has_voted_detail(N, B, R, Bl) := false;
       node_has_voted(N,B) := false;
       node_has_locked_detail(N, B, R) := false;
       node_has_locked(N,B) := false;
       node_sent_timeout(N, T) := false;
       node_has_processed_qc(N,Q) := false;
       quorum_of_votes(Bc, Bp) := false;
       quorum_of_recent_locks(B,R) := false;
       quorum_of_recent_locks(block_t.nil,R) := true;

assume forall B:block_t. B.parent = block_t.hash(block_t.nil) if B ~=
    block_t.nil else hash_t.nil;

assume forall B:block_t. B.round = round_t.next(0) if B ~= block_t.nil
    else 0;
}

before node_voted(n:node_t, b:block_t, r:round_t, bl:block_t) {
       require is_good(n) -> (forall B:block_t. (node_has_voted(n,B) &
```

```
            is_good(n)) -> b.round > B.round);
        require is_good(n) -> b.parent = block_t.hash(bl);
        require (is_good(n) & bl ~= block_t.nil) ->
            node_has_locked_recently(n,bl,r);
        require (is_good(n) & bl = block_t.nil) -> (
            node_has_locked_detail(n,B,R) -> B = block_t.nil);
        require is_good(n) -> round_t.succ(b.round,r);
}

before node_locked_quorum(n:node_t, b:block_t, q:quorum_t, r:round_t) {
        require forall N:node_t. quorum_t.member(N, q) -> node_has_voted(
            N, b);
        require ~is_good(n) | forall Rs:round_t. forall B1,B2:block_t. r
            < Rs -> ~node_has_voted_detail(n,B1,Rs,B2);
        require ~is_good(n) | b.round < r;
        require ~is_good(n) | forall B:block_t. forall R:round_t.
            node_has_locked_detail(n,B,R) -> B.round < b.round
}

before node_timeout(n:node_t, t:timeout_t) {
        require (t.qc.block ~= block_t.nil & is_good(n)) ->
            node_has_locked(n, t.qc.block);
}

before node_processed_qc(n:node_t, qc:cert_t) {
        require qc.block ~= block_t.nil;
        require forall N:node_t. quorum_t.member(N, qc.cert) ->
            node_has_voted(N, qc.block);
}

implement node_voted(n:node_t, b:block_t, r:round_t, bl:block_t) {
        node_has_voted_detail(n, b, r, bl) := true;
        node_has_voted(n,b) := true;

        if (exists Q:quorum_t. exists R:round_t. round_t.succ(b.round,R)
            & b.parent=block_t.hash(bl) & forall N1:node_t. ((quorum_t.
            member(N1,Q) & is_good(N1)) -> (gv.node_has_voted_detail(N1,b
            ,R,bl))))
        {
                quorum_of_votes(b,bl) := true;
        }
}

implement node_locked_quorum(n:node_t, b:block_t, q:quorum_t, r:round_t)
     {
        node_has_locked_detail(n, b, r) := true;
        node_has_locked(n,b) := true;
        quorum_of_recent_locks(B,R) := exists Q:quorum_t. forall N:node_t
            . ((quorum_t.member(N,Q) & is_good(N) & B~= block_t.nil) ->
```

```
                node_has_locked_recently(N,B,R))
        }

        implement node_timeout(n:node_t, t:timeout_t) {
                node_sent_timeout(n, t) := true;
        }

        implement node_processed_qc(n:node_t, qc:cert_t) {
                node_has_processed_qc(n,qc) := true;
        }
}
```

B.6. classic_safety.ivy

```
#lang ivy1.8

include domain_model
include network_model
include algorithm
include global_view

isolate gv = {
        axiom forall R:round_t. R = R

        instantiate global_view

        invariant forall N:node_t. forall B:block_t. forall R:round_t. forall Bl
            :block_t. (node_has_voted_detail(N,B,R,Bl) & is_good(N)) -> (round_t
            .succ(B.round,R) & B.parent = block_t.hash(Bl) & node_has_voted(N,B)
            )

        invariant forall N:node_t. forall B:block_t. (is_good(N) &
            node_has_locked(N,B)) -> exists Q:quorum_t. forall N1:node_t. (
            quorum_t.member(N1,Q) -> node_has_voted(N1,B))

        invariant forall B1:block_t. forall B2:block_t. forall N:node_t. (
            is_good(N) & node_has_voted(N,B1) & node_has_voted(N,B2) & B1.round
            = B2.round) -> B1 = B2

        invariant forall N1:node_t. forall N2:node_t. forall B1:block_t. forall
            B2:block_t. (is_good(N1) & is_good(N2) & node_has_locked(N1, B1) &
            node_has_locked(N2, B2) & B1.round = B2.round) -> B1 = B2

        invariant forall N:node_t. forall Qc:cert_t. (is_good(N) &
            node_has_processed_qc(N,Qc)) -> exists Q:quorum_t. forall N1:node_t.
             (quorum_t.member(N1,Q) -> node_has_voted(N1,Qc.block))
```

```
        invariant forall N1:node_t. forall N2:node_t. forall Q1:cert_t. forall
            Q2:cert_t. (is_good(N1) & is_good(N2) & node_has_processed_qc(N1, Q1
            ) & node_has_processed_qc(N2, Q2) & Q1.block.round = Q2.block.round)
             -> Q1.block = Q2.block

        invariant forall N:node_t. forall B:block_t. forall R:round_t. (is_good(
            N) & node_has_voted_detail(N,B,R,block_t.nil)) -> (forall Rp:round_t
            . forall Bgdc:block_t. (Rp < R & node_has_locked_detail(N,Bgdc,Rp))
            -> Bgdc = block_t.nil)

        invariant forall N:node_t. forall Qc:cert_t. (is_good(N) &
            node_has_processed_qc(N,Qc)) -> Qc.block ~= block_t.nil
} with quorum_t, round_t, block_t

isolate gdc_properties = {
        relation gdc(B:block_t)

        definition gdc(B:block_t) = exists Q:quorum_t. exists R:round_t. (
            round_t.succ(B.round,R) & (forall N:node_t. (quorum_t.member(N,Q) &
            is_good(N)) -> gv.node_has_locked_detail(N,B,R)))

        invariant forall Bc:block_t. forall Bp:block_t. Bc.parent = block_t.hash
            (Bp) -> Bp.round < Bc.round

        invariant forall B:block_t. block_t.nil.round <= B.round
} with gv,round_t, block_t

isolate gdc_properties1 = {
        invariant forall N:node_t. forall B:block_t. forall R:round_t. forall Bl
            :block_t. (gv.node_has_voted_detail(N,B,R,Bl) & is_good(N)) -> (
            round_t.succ(B.round,R) & B.parent = block_t.hash(Bl) & gv.
            node_has_voted(N,B) & (Bl = block_t.nil | gv.
            node_has_locked_recently(N,Bl,R)))

        invariant forall N:node_t. forall B:block_t. forall R:round_t. (is_good(
            N) & gv.node_has_locked_detail(N,B,R)) -> B.round < R

        private {
               invariant forall N:node_t. forall B:block_t. (is_good(N) & gv.
                   node_has_voted(N,B)) -> exists R:round_t. exists Bp:block_t.
                   (gv.node_has_voted_detail(N,B,R,Bp) & round_t.succ(B.round,R)
                    & B.parent = block_t.hash(Bp))
        }

        invariant forall Bc:block_t. forall Bp:block_t. gv.quorum_of_votes(Bc,Bp
            ) <-> exists Q:quorum_t. exists R:round_t. round_t.succ(Bc.round,R)
            & Bc.parent=block_t.hash(Bp) & forall N1:node_t. ((quorum_t.member(
            N1,Q) & is_good(N1)) -> (gv.node_has_voted_detail(N1,Bc,R,Bp) &
            is_good(N1)))
```

```
        invariant forall Bc:block_t. forall Bp:block_t. forall N:node_t. (
            is_good(N) & gv.node_has_locked(N,Bc) & Bc.parent=block_t.hash(Bp) &
             Bp ~= block_t.nil) -> gv.quorum_of_votes(Bc,Bp)

        invariant forall Bc:block_t. forall Bp:block_t. forall N:node_t. (
            is_good(N) & gv.node_has_locked(N,Bc) & Bc.parent=block_t.hash(Bp))
            -> gv.quorum_of_votes(Bc,Bp)

        invariant forall Bp:block_t. forall R:round_t. (gv.
            quorum_of_recent_locks(Bp,R) <-> exists Q:quorum_t. forall N:node_t.
             (quorum_t.member(N,Q) & is_good(N) & Bp~= block_t.nil) -> gv.
            node_has_locked_recently(N,Bp,R))
} with gv, quorum_t, round_t, block_t

isolate gdc_properties2 = {
        private {
                invariant forall N:node_t. forall B:block_t. forall R:round_t.
                    forall Bl:block_t. (gv.node_has_voted_detail(N,B,R,Bl) &
                    is_good(N)) -> (round_t.succ(B.round,R) & B.parent = block_t.
                    hash(Bl) & gv.node_has_voted(N,B) & (Bl = block_t.nil | gv.
                    node_has_locked_recently(N,Bl,R)))

                invariant forall N:node_t. forall B:block_t. forall R:round_t.
                    forall Bl:block_t. (gv.node_has_voted_detail(N,B,R,Bl) &
                    is_good(N) & Bl ~= block_t.nil) -> (round_t.succ(B.round,R) &
                     B.parent = block_t.hash(Bl) & gv.node_has_voted(N,B) & gv.
                    node_has_locked_recently(N,Bl,R))
        }

        property forall R1,R2,R3:round_t. (round_t.succ(R1,R2) & round_t.succ(R1
            ,R3)) -> R2 = R3

        invariant forall Bc:block_t. forall Bp:block_t. forall R:round_t. (gv.
            quorum_of_votes(Bc,Bp) & round_t.succ(Bc.round,R) & Bp ~= block_t.
            nil) -> Bp ~= block_t.nil & exists Q:quorum_t. (round_t.succ(Bc.
            round,R) & Bc.parent=block_t.hash(Bp) & forall N1:node_t. ((quorum_t
            .member(N1,Q) & is_good(N1)) -> (gv.node_has_voted_detail(N1,Bc,R,Bp
            ))))

        property forall Bc:block_t. forall Bp:block_t. forall R:round_t. (Bp ~=
            block_t.nil & (exists Q1:quorum_t. forall N1:node_t. (is_good(N1) &
            quorum_t.member(N1,Q1)) -> gv.node_has_voted_detail(N1,Bc,R,Bp)) & (
            forall N:node_t. (is_good(N) & gv.node_has_voted_detail(N,Bc,R,Bp))
            -> gv.node_has_locked_recently(N,Bp,R))) -> (exists Q2:quorum_t.
            forall N2:node_t. (is_good(N2) & quorum_t.member(N2,Q2)) -> gv.
            node_has_locked_recently(N2,Bp,R))

        invariant forall Bc:block_t. forall Bp:block_t. forall R:round_t. (Bp ~=
```

```
            block_t.nil & (exists Q1:quorum_t. forall N1:node_t. (is_good(N1) &
            quorum_t.member(N1,Q1)) -> gv.node_has_voted_detail(N1,Bc,R,Bp)) &
           (forall N:node_t. (is_good(N) & gv.node_has_voted_detail(N,Bc,R,Bp))
            -> gv.node_has_locked_recently(N,Bp,R))) -> (exists Q2:quorum_t.
           forall N2:node_t. (is_good(N2) & quorum_t.member(N2,Q2)) -> gv.
           node_has_locked_recently(N2,Bp,R))

       invariant forall Bc:block_t. forall Bp:block_t. forall R:round_t. (gv.
           quorum_of_votes(Bc,Bp) & round_t.succ(Bc.round,R) & Bp ~= block_t.
           nil) -> exists Q2:quorum_t. forall N2:node_t. (is_good(N2) &
           quorum_t.member(N2,Q2)) -> gv.node_has_locked_recently(N2,Bp,R)
} with gv, quorum_t, round_t, gdc_properties,gdc_properties1,block_t

isolate gdc_properties2a = {
       invariant forall Bc:block_t. forall Bp:block_t. forall R:round_t. (gv.
           quorum_of_votes(Bc,Bp) & round_t.succ(Bc.round,R) & Bp ~= block_t.
           nil) -> gv.quorum_of_recent_locks(Bp,R)

       invariant forall N:node_t. forall B:block_t. forall R:round_t. (is_good(
           N) & gv.node_has_locked_detail(N,B,R)) -> B.round < R

       invariant forall Bc:block_t. forall Bp:block_t. forall R:round_t. forall
            N:node_t. (round_t.succ(Bp.round,Bc.round) & round_t.succ(Bp.round,
           R) & Bc.parent = block_t.hash(Bp) & gv.node_has_locked_recently(N,Bp
           ,R) & is_good(N)) -> gv.node_has_locked_detail(N,Bp,Bc.round)

       invariant forall B:block_t. forall Bp:block_t. (exists N:node_t. (
           is_good(N) & gv.node_has_locked(N,B)) & B.parent=block_t.hash(Bp) &
           round_t.succ(Bp.round,B.round) & Bp ~= block_t.nil) ->
           gdc_properties.gdc(Bp)
} with gv, quorum_t, round_t, gdc_properties,gdc_properties1,block_t,
    gdc_properties2

isolate gdc_properties3 = {
       invariant forall R1,R2,R3:round_t. (round_t.succ(R1,R2) & round_t.succ(
           R1,R3)) -> R2 = R3

       private {
              invariant forall N:node_t. forall B:block_t. forall R:round_t.
                  forall Bp:block_t. (is_good(N) & gv.node_has_voted(N,B) &
                  round_t.succ(B.round,R) & B.parent = block_t.hash(Bp)) -> gv.
                  node_has_voted_detail(N,B,R,Bp)
       }

       invariant forall N:node_t. forall Bc:block_t. forall R:round_t. forall
           Bp:block_t. forall Qc:cert_t. (is_good(N) & gv.node_has_processed_qc
           (N,Qc) & Qc.block = Bc & Bc.parent=block_t.hash(Bp) & Bp ~= block_t.
           nil & round_t.succ(Bc.round,R)) -> forall N1:node_t. (is_good(N1) &
           quorum_t.member(N1,Qc.cert)) -> (is_good(N1) & gv.node_has_voted(N1,
```

```
            Bc) & round_t.succ(Bc.round,R) & Bc.parent=block_t.hash(Bp))

        invariant forall N:node_t. forall Bc:block_t. forall R:round_t. forall
            Bp:block_t. forall Qc:cert_t. (is_good(N) & gv.node_has_processed_qc
            (N,Qc) & Qc.block = Bc & Bc.parent=block_t.hash(Bp) & Bp ~= block_t.
            nil & round_t.succ(Bc.round,R)) -> forall N1:node_t. (is_good(N1) &
            quorum_t.member(N1,Qc.cert)) -> (is_good(N1) & gv.
            node_has_voted_detail(N1,Bc,R,Bp))

        invariant forall N:node_t. forall B:block_t. forall R:round_t. forall Bl
            :block_t. (gv.node_has_voted_detail(N,B,R,Bl) & is_good(N)) -> (
            round_t.succ(B.round,R) & B.parent = block_t.hash(Bl) & gv.
            node_has_voted(N,B) & (Bl = block_t.nil | gv.
            node_has_locked_recently(N,Bl,R)))

        invariant forall N:node_t. forall Bc:block_t. forall R:round_t. forall
            Bp:block_t. forall Qc:cert_t. (is_good(N) & gv.node_has_processed_qc
            (N,Qc) & Qc.block = Bc & Bc.parent=block_t.hash(Bp) & Bp ~= block_t.
            nil & round_t.succ(Bc.round,R)) -> forall N1:node_t. (is_good(N1) &
            quorum_t.member(N1,Qc.cert)) -> (is_good(N1) & (Bp ~= block_t.nil |
            gv.node_has_locked_recently(N1,Bp,R)))
} with gv, quorum_t, round_t, gdc_properties,gdc_properties1, gdc_properties2,
    gdc_properties2a, block_t

isolate gdc_properties4 = {
        invariant forall Bgdc:block_t. forall Rgdc:round_t. (gdc_properties.gdc(
            Bgdc) & round_t.succ(Bgdc.round,Rgdc)) -> exists Q:quorum_t. (forall
             N:node_t. (quorum_t.member(N,Q) & is_good(N)) -> gv.
            node_has_locked_detail(N,Bgdc,Rgdc))

        invariant forall Bgdc:block_t. forall Rgdc:round_t. (gdc_properties.gdc(
            Bgdc) & round_t.succ(Bgdc.round,Rgdc)) -> exists N:node_t. (is_good(
            N) & gv.node_has_locked_detail(N,Bgdc,Rgdc))
} with gv, quorum_t, round_t, block_t, gdc_properties,gdc_properties1

isolate gdc_properties5 = {
        relation quorum_of_locks(B:block_t, R:round_t)

        definition quorum_of_locks(B:block_t, R:round_t) = exists Q:quorum_t.
            forall N:node_t. (is_good(N) & quorum_t.member(N,Q)) -> gv.
            node_has_locked_detail(N,B,R)

        invariant forall Bp:block_t. forall R:round_t. (gv.
            quorum_of_recent_locks(Bp,R) & Bp ~= block_t.nil) -> exists Q:
            quorum_t. forall N:node_t. (quorum_t.member(N,Q) & is_good(N)) -> gv
            .node_has_locked_recently(N,Bp,R)

        invariant forall B:block_t. forall R:round_t. quorum_of_locks(B, R) ->
            exists Q:quorum_t. forall N:node_t. (is_good(N) & quorum_t.member(N,
```

```
            Q)) -> gv.node_has_locked_detail(N,B,R)

        invariant forall Bgdc:block_t. forall Rgdc:round_t. forall Blr:block_t.
            forall Rlr:round_t. (quorum_of_locks(Bgdc,Rgdc) & gv.
            quorum_of_recent_locks(Blr,Rlr) & Blr~= block_t.nil) -> exists N:
            node_t. (is_good(N) & gv.node_has_locked_detail(N,Bgdc,Rgdc) & gv.
            node_has_locked_recently(N,Blr,Rlr))

        invariant forall Bgdc:block_t. forall Blr:block_t. exists Rgdc:round_t.
            exists Rlr:round_t. (quorum_of_locks(Bgdc,Rgdc) & gv.
            quorum_of_recent_locks(Blr,Rlr) & Blr~= block_t.nil & Rgdc < Rlr) ->
             exists N:node_t. (is_good(N) & gv.node_has_locked_detail(N,Bgdc,
            Rgdc) & gv.node_has_locked_recently(N,Blr,Rlr) & Rgdc < Rlr)
} with gv,quorum_t

isolate gdc_supplementary_defs = {
        relation no_lock(N:node_t,Rl:round_t,Ru:round_t,Bs:block_t)

        definition no_lock(N,Rl,Ru,Bs:block_t) = forall R:round_t. forall B:
            block_t. (((Rl < R & R < Ru) -> ~gv.node_has_locked_detail(N,B,R)) &
             ((gv.node_has_locked_detail(N,B,Rl) & B ~= Bs) -> B.round < Bs.
            round))

        relation node_has_two_locks(Bc:block_t, Bgdc:block_t, Bp:block_t)

        definition node_has_two_locks(Bc:block_t, Bgdc:block_t, Bp:block_t) =
            exists N1:node_t. exists Rcp1:round_t. exists Rgdc:round_t. is_good(
            N1) &round_t.succ(Bc.round,Rcp1) & round_t.succ(Bgdc.round,Rgdc) &
            gv.node_has_locked_detail(N1,Bgdc,Rgdc) & gv.
            node_has_locked_recently(N1,Bp,Rcp1) & Rgdc < Rcp1 & Bc.parent=
            block_t.hash(Bp)

        relation node_has_recent_lock(Bc:block_t,Bgdc:block_t,Bp:block_t)

        definition node_has_recent_lock(Bc:block_t,Bgdc:block_t,Bp:block_t) =
            exists N1:node_t. exists Rcp1:round_t. exists Rl:round_t. exists
            Rgdc:round_t. is_good(N1) & round_t.succ(Bc.round,Rcp1) & round_t.
            succ(Bgdc.round,Rgdc) & Rgdc < Rcp1 & Bc.parent=block_t.hash(Bp) &
            Rgdc <= Rl & Rl < Rcp1 & gv.node_has_locked_detail(N1,Bp,Rl)& gv.
            node_has_locked_detail(N1,Bgdc,Rgdc) & no_lock(N1,Rl,Rcp1,Bp)
}

isolate gdc_properties6 = {
        private {
                invariant forall N:node_t. forall Bs:block_t. forall Rs:round_t.
                    gv.node_has_locked_recently(N,Bs,Rs) -> exists Rl:round_t. (
                    Rl < Rs & gv.node_has_locked_detail(N,Bs,Rl) &
                    gdc_supplementary_defs.no_lock(N,Rl,Rs,Bs))
        }
```

```
        invariant forall N:node_t. forall Bs:block_t. forall Rs:round_t. forall
            Bgdc:block_t. forall Rgdc:round_t. (gv.node_has_locked_recently(N,Bs
            ,Rs) & gv.node_has_locked_detail(N,Bgdc,Rgdc) & Rgdc < Rs) -> (
            forall Rl:round_t. Rl < Rgdc -> ~gdc_supplementary_defs.no_lock(N,Rl
            ,Rs,Bs))

        invariant forall Bc:block_t. forall Bgdc:block_t. forall Bp:block_t. (
            gdc_supplementary_defs.node_has_two_locks(Bc,Bgdc,Bp) ->
            gdc_supplementary_defs.node_has_recent_lock(Bc,Bgdc,Bp))
} with gv, quorum_t, round_t, block_t,gdc_supplementary_defs

isolate gdc_properties7m1 = {
        invariant forall Bc:block_t. forall Bp:block_t. forall N:node_t. (
            is_good(N) & gv.node_has_locked(N,Bc) & Bc.parent=block_t.hash(Bp) &
             Bp ~= block_t.nil) -> gv.quorum_of_votes(Bc,Bp)
} with gv,round_t,gdc_properties1

isolate gdc_properties7m2 = {
        invariant forall Bc:block_t. forall Bp:block_t. (gv.quorum_of_votes(Bc,
            Bp) & Bp ~= block_t.nil) -> (exists R:round_t. round_t.succ(Bc.round
            ,R) & Bp ~= block_t.nil & gv.quorum_of_recent_locks(Bp,R))
} with gv,round_t,quorum_t,gdc_properties2,gdc_properties2a


isolate gdc_properties7 = {
        invariant forall Bc:block_t. forall Bp:block_t. forall N:node_t. (
            is_good(N) & gv.node_has_locked(N,Bc) & Bc.parent=block_t.hash(Bp) &
             Bp ~= block_t.nil) -> exists Rcp1:round_t. round_t.succ(Bc.round,
            Rcp1) & gv.quorum_of_recent_locks(Bp,Rcp1)
} with gv,round_t,gdc_properties7m1,gdc_properties7m2

isolate gdc_properties8 = {
        invariant forall Bgdc:block_t. gdc_properties.gdc(Bgdc) -> exists Rgdc:
            round_t. round_t.succ(Bgdc.round,Rgdc) & gdc_properties5.
            quorum_of_locks(Bgdc,Rgdc)
} with gv,round_t,gdc_properties5,quorum_t,gdc_properties

isolate gdc_properties9 = {
        invariant forall Bc:block_t. forall Bp:block_t. forall N:node_t. forall
            Bgdc:block_t. (is_good(N) & gv.node_has_locked(N,Bc) & Bc.parent=
            block_t.hash(Bp) & Bp ~= block_t.nil & gdc_properties.gdc(Bgdc) &
            Bgdc.round < Bc.round) -> (exists Rcp1:round_t. round_t.succ(Bc.
            round,Rcp1) & gv.quorum_of_recent_locks(Bp,Rcp1) & exists Rgdc:
            round_t. round_t.succ(Bgdc.round,Rgdc) & gdc_properties5.
            quorum_of_locks(Bgdc,Rgdc) & Rgdc < Rcp1)
} with gv, round_t,gdc_properties7,gdc_properties8

isolate gdc_properties10 = {
```

```
        relation gdc_and_lock(Bc:block_t,Bp:block_t,N:node_t,Bgdc:block_t)

        definition gdc_and_lock(Bc:block_t,Bp:block_t,N:node_t,Bgdc:block_t) =
            is_good(N) & gv.node_has_locked(N,Bc) & Bc.parent=block_t.hash(Bp) &
             Bp ~= block_t.nil & gdc_properties.gdc(Bgdc) & Bgdc.round < Bc.
            round

        invariant forall Bc:block_t. forall Bp:block_t. forall N:node_t. forall
            Bgdc:block_t. gdc_and_lock(Bc,Bp,N,Bgdc) -> gdc_supplementary_defs.
            node_has_two_locks(Bc,Bgdc,Bp)
} with gv, round_t,gdc_properties5,gdc_properties9,gdc_supplementary_defs

isolate gdc_properties11 = {
        invariant forall Bc:block_t. forall Bp:block_t. forall N:node_t. forall
            Bgdc:block_t. gdc_properties10.gdc_and_lock(Bc,Bp,N,Bgdc) ->
            gdc_supplementary_defs.node_has_recent_lock(Bc,Bgdc,Bp)
} with gv,gdc_properties10,gdc_properties6

isolate gdc_properties9am1 = {
        invariant forall Bc:block_t. forall Bp:block_t. (gv.quorum_of_votes(Bc,
            Bp) & Bc.parent=block_t.hash(Bp) & Bp ~= block_t.nil) -> (exists
            Rcp1:round_t. round_t.succ(Bc.round,Rcp1) & gv.
            quorum_of_recent_locks(Bp,Rcp1))
} with gdc_properties2,gdc_properties2a,gv, round_t

isolate gdc_properties9a = {
        private {
                invariant forall Bc:block_t. forall Bp:block_t. forall Rcp1:
                    round_t. forall Bgdc:block_t. (round_t.succ(Bc.round,Rcp1) &
                    gv.quorum_of_votes(Bc,Bp) & Bc.parent=block_t.hash(Bp) & Bp
                    ~= block_t.nil & gdc_properties.gdc(Bgdc) & Bgdc.round < Bc.
                    round) -> (gv.quorum_of_recent_locks(Bp,Rcp1) & exists Rgdc:
                    round_t. round_t.succ(Bgdc.round,Rgdc) & gdc_properties5.
                    quorum_of_locks(Bgdc,Rgdc) & Rgdc < Rcp1)
        }
        invariant forall Bc:block_t. forall Bp:block_t. forall N:node_t. forall
            Bgdc:block_t. (is_good(N) & gv.quorum_of_votes(Bc,Bp) & Bc.parent=
            block_t.hash(Bp) & Bp ~= block_t.nil & gdc_properties.gdc(Bgdc) &
            Bgdc.round < Bc.round) -> (exists Rcp1:round_t. round_t.succ(Bc.
            round,Rcp1) & gv.quorum_of_recent_locks(Bp,Rcp1) & exists Rgdc:
            round_t. round_t.succ(Bgdc.round,Rgdc) & gdc_properties5.
            quorum_of_locks(Bgdc,Rgdc) & Rgdc < Rcp1)

        invariant forall Bc:block_t. forall Bp:block_t. forall Bgdc:block_t. (gv
            .quorum_of_votes(Bc,Bp) & Bc.parent=block_t.hash(Bp) & Bp ~= block_t
            .nil & gdc_properties.gdc(Bgdc) & Bgdc.round < Bc.round) -> (exists
            Rcp1:round_t. round_t.succ(Bc.round,Rcp1) & gv.
            quorum_of_recent_locks(Bp,Rcp1) & exists Rgdc:round_t. round_t.succ(
            Bgdc.round,Rgdc) & gdc_properties5.quorum_of_locks(Bgdc,Rgdc) & Rgdc
```

```
            < Rcp1)
} with gv, round_t,gdc_properties9am1,gdc_properties8

isolate gdc_properties10a = {
       relation gdc_and_quorum(Bc:block_t,Bp:block_t,Bgdc:block_t)

       definition gdc_and_quorum(Bc:block_t,Bp:block_t,Bgdc:block_t) = gv.
           quorum_of_votes(Bc,Bp) & Bc.parent=block_t.hash(Bp) & Bp ~= block_t.
           nil & gdc_properties.gdc(Bgdc) & Bgdc.round < Bc.round

       invariant forall Bc:block_t. forall Bp:block_t. forall Bgdc:block_t.
           gdc_and_quorum(Bc,Bp,Bgdc) -> gdc_supplementary_defs.
           node_has_two_locks(Bc,Bgdc,Bp)
} with gv, round_t,gdc_properties5,gdc_properties9a,gdc_supplementary_defs

isolate gdc_properties11a = {
       invariant forall Bc:block_t. forall Bp:block_t. forall Bgdc:block_t.
           gdc_properties10a.gdc_and_quorum(Bc,Bp,Bgdc) ->
           gdc_supplementary_defs.node_has_recent_lock(Bc,Bgdc,Bp)
} with gv,gdc_properties10a,gdc_properties6


isolate quorum_propagationm1 = {
       invariant forall Bc:block_t. forall Bp:block_t. forall R:round_t. (gv.
           quorum_of_votes(Bc,Bp) & round_t.succ(Bc.round,R) & Bp ~= block_t.
           nil) -> gv.quorum_of_recent_locks(Bp,R)

} with gv,gdc_properties2,gdc_properties2a

isolate quorum_propagationm2 = {
       private {
              invariant forall N:node_t. forall B:block_t. (is_good(N) & gv.
                  node_has_voted(N,B)) -> exists R:round_t. exists Bp:block_t.
                  (gv.node_has_voted_detail(N,B,R,Bp) & round_t.succ(B.round,R)
                   & B.parent = block_t.hash(Bp))
       }

       invariant forall Bc:block_t. forall Bp:block_t. (Bc.parent = block_t.
           hash(Bp)) -> Bp.round < Bc.round

       invariant forall B:block_t. block_t.nil.round <= B.round

       invariant forall Bc:block_t. forall Bp:block_t. gv.quorum_of_votes(Bc,Bp
           ) <-> exists Q:quorum_t. exists R:round_t. round_t.succ(Bc.round,R)
           & Bc.parent=block_t.hash(Bp) & forall N1:node_t. ((quorum_t.member(
           N1,Q) & is_good(N1)) -> (gv.node_has_voted_detail(N1,Bc,R,Bp) &
           is_good(N1)))

       invariant forall B:block_t. forall Bp1:block_t. forall Bp2:block_t. (B.
```

```
            parent = block_t.hash(Bp1) & B.parent = block_t.hash(Bp2)) -> Bp1 =
            Bp2

        invariant forall N:node_t. forall B:block_t. forall R:round_t. gv.
            node_has_locked_detail(N,B,R) -> gv.node_has_locked(N,B)

        invariant forall Bp:block_t. forall Bgp:block_t. forall N:node_t. (
            is_good(N) & gv.node_has_locked(N,Bp) & Bp.parent=block_t.hash(Bgp))
             -> gv.quorum_of_votes(Bp,Bgp)
} with gv,quorum_t,block_t,round_t,quorum_propagationm1

isolate quorum_propagation = {
        relation block_has_qov(B:block_t)

        definition block_has_qov(B:block_t) = B ~= block_t.nil & (exists Bp:
            block_t. (B.parent=block_t.hash(Bp) & gv.quorum_of_votes(B,Bp)))

        private {
                invariant forall Bp:block_t. forall R:round_t. (gv.
                    quorum_of_recent_locks(Bp,R) & Bp ~= block_t.nil) -> exists N
                    :node_t. (is_good(N) & gv.node_has_locked(N,Bp))

                invariant forall N:node_t. forall B:block_t. (is_good(N) & gv.
                    node_has_voted(N,B)) -> exists Bp:block_t. ( B.parent =
                    block_t.hash(Bp))
        }

        invariant forall B:block_t. forall Bp:block_t. (block_has_qov(B) & B.
            parent = block_t.hash(Bp) & Bp ~= block_t.nil) -> block_has_qov(Bp)
} with gv,quorum_t,block_t,quorum_propagationm1,quorum_propagationm2

isolate gdc_properties12 = {
        relation nil_locks_upto_round(N:node_t, R:round_t)

        definition nil_locks_upto_round(N:node_t, R:round_t) = forall Rp:round_t
            . forall B:block_t. (Rp <= R & gv.node_has_locked_detail(N,B,Rp) ->
            B=block_t.nil)

        invariant forall Bc:block_t. (quorum_propagation.block_has_qov(Bc) & Bc.
            parent = block_t.hash(block_t.nil)) -> gv.quorum_of_votes(Bc,block_t
            .nil)

        invariant forall Bc:block_t. gv.quorum_of_votes(Bc,block_t.nil) ->
            exists Q:quorum_t. forall N:node_t. ((is_good(N) & quorum_t.member(N
            ,Q)) -> nil_locks_upto_round(N,Bc.round))

        invariant forall Bc:block_t. (quorum_propagation.block_has_qov(Bc) & Bc.
            parent = block_t.hash(block_t.nil)) -> exists Q:quorum_t. forall N:
            node_t. ((is_good(N) & quorum_t.member(N,Q)) -> nil_locks_upto_round
```

```
            (N,Bc.round))
} with gv,round_t,block_t,quorum_propagation,gdc_properties1

isolate gdc_properties13 = {
        invariant forall Bgdc:block_t. forall Bc:block_t. ((exists Q1:quorum_t.
            forall N:node_t. (quorum_t.member(N,Q1) & is_good(N)) ->
            gdc_properties12.nil_locks_upto_round(N,Bc.round))) & (exists Q2:
            quorum_t. exists R:round_t. (round_t.succ(Bgdc.round,R) & (forall N:
            node_t. (quorum_t.member(N,Q2) & is_good(N)) -> gv.
            node_has_locked_detail(N,Bgdc,R)))) -> exists N:node_t. (is_good(N)
            & quorum_t.member(N,Q1) & quorum_t.member(N,Q2))

        invariant forall Bgdc:block_t. forall Bc:block_t. (((exists Q1:quorum_t.
             forall N:node_t. (quorum_t.member(N,Q1) & is_good(N)) ->
            gdc_properties12.nil_locks_upto_round(N,Bc.round))) & (exists Q2:
            quorum_t. exists R:round_t. (round_t.succ(Bgdc.round,R) & (forall N:
            node_t. (quorum_t.member(N,Q2) & is_good(N)) -> gv.
            node_has_locked_detail(N,Bgdc,R)))) & Bgdc ~= block_t.nil)-> (exists
             N:node_t. is_good(N) & gdc_properties12.nil_locks_upto_round(N,Bc.
            round) & (exists R:round_t. (round_t.succ(Bgdc.round,R) & gv.
            node_has_locked_detail(N,Bgdc,R))) & Bgdc ~= block_t.nil)

        invariant forall Bgdc:block_t. forall Bc:block_t. (exists N:node_t. (
            is_good(N) & gdc_properties12.nil_locks_upto_round(N,Bc.round) & (
            exists R:round_t. (round_t.succ(Bgdc.round,R) & gv.
            node_has_locked_detail(N,Bgdc,R))) & Bgdc ~= block_t.nil)) -> Bc.
            round <= Bgdc.round

        invariant forall Bgdc:block_t. forall Bc:block_t. ((exists Q1:quorum_t.
            forall N:node_t. (quorum_t.member(N,Q1) & is_good(N)) ->
            gdc_properties12.nil_locks_upto_round(N,Bc.round)) & (exists Q2:
            quorum_t. exists R:round_t. (round_t.succ(Bgdc.round,R) & (forall N:
            node_t. (quorum_t.member(N,Q2) & is_good(N)) -> gv.
            node_has_locked_detail(N,Bgdc,R)))) & Bgdc ~= block_t.nil) -> Bc.
            round <= Bgdc.round
} with gv,round_t,block_t,quorum_t,gdc_properties12,gdc_properties,
    quorum_propagation

isolate gdc_properties14 = {
        invariant forall Bgdc:block_t. forall Bc:block_t. (quorum_propagation.
            block_has_qov(Bc) & Bc.parent = block_t.hash(block_t.nil) &
            gdc_properties.gdc(Bgdc) & Bgdc ~= block_t.nil) -> Bc.round <= Bgdc.
            round
} with gv,round_t,block_t,quorum_t,gdc_properties,gdc_properties12,
    gdc_properties13

isolate gdc_properties15 = {
        invariant forall Bgdc:block_t. forall Bp:block_t. (gdc_properties.gdc(
            Bgdc) & Bgdc.parent=block_t.hash(Bp) & Bp ~= block_t.nil) -> exists
```

```
            N:node_t. (is_good(N) & gv.node_has_locked(N,Bgdc) & gv.
            quorum_of_votes(Bgdc,Bp))

        invariant forall Bgdc:block_t. forall Bp:block_t. (gdc_properties.gdc(
            Bgdc) & Bgdc.parent=block_t.hash(Bp)) -> exists N:node_t. (is_good(N
            ) & gv.node_has_locked(N,Bgdc) & gv.quorum_of_votes(Bgdc,Bp))
} with gv, quorum_t, round_t, block_t,gdc_properties,gdc_properties1

isolate gdc_properties16 = {
        invariant forall Bgdc:block_t. forall Bp:block_t. (Bgdc.parent=block_t.
            hash(Bp) & gv.quorum_of_votes(Bgdc,Bp)) -> quorum_propagation.
            block_has_qov(Bgdc)

        private {
                invariant forall Bgdc:block_t. forall Bp:block_t. (gdc_properties
                    .gdc(Bgdc) & Bgdc.parent=block_t.hash(Bp)) -> exists N:node_t
                    . (is_good(N) & gv.node_has_locked(N,Bgdc) & gv.
                    quorum_of_votes(Bgdc,Bp))
        }

        invariant forall Bgdc:block_t. forall Bp:block_t. (gdc_properties.gdc(
            Bgdc) & Bgdc.parent=block_t.hash(Bp)) -> quorum_propagation.
            block_has_qov(Bgdc)
} with gv, quorum_t, round_t, block_t,gdc_properties,gdc_properties15,
    quorum_propagation

isolate basic_safety = {
        isolate processor = {
                instantiate supraBFT(gv)

                invariant forall Bc:block_t. forall Bp:block_t. Bc.parent =
                    block_t.hash(Bp) -> Bp.round < Bc.round

                invariant forall B:block_t. block_t.nil.round <= B.round

                invariant forall M:msg. forall D:node_t. (shim.sent(M,D) & M.kind
                     = msg_kind.prepare) -> gv.node_has_voted(M.src,M.prep)

                invariant forall M:msg. forall D:node_t. (shim.sent(M,D) & M.kind
                    =msg_kind.timeout) -> gv.node_sent_timeout(M.src,M.t)

                invariant forall N:node_t. forall H:height_t. (is_good(N) &
                    validator(N).blockchain(H) ~= block_t.nil) -> exists Q:cert_t
                    . (Q.block = validator(N).blockchain(H) & validator(N).
                    qc_processed(Q))

                invariant forall N:node_t. forall B:block_t. forall R:round_t. (
                    is_good(N) & gv.node_has_locked_detail(N,B,R)) -> (B.round <
                    validator(N).locked.block.round | B = validator(N).locked.
```

```
            block)

        invariant forall N:node_t. forall B:block_t. forall Rl:round_t.
            forall Rs:round_t. (is_good(N) & gv.node_has_locked_detail(N,
            validator(N).locked.block,Rl) & Rl < Rs) -> ~gv.
            node_has_locked_detail(N,B,Rs)

        invariant forall N:node_t. forall B:block_t. forall R:round_t.
            forall Bl:block_t. (is_good(N) & gv.node_has_voted_detail(N,B
            ,R,Bl)) -> R <= validator(N).r_c

        invariant forall N:node_t. forall B:block_t. forall R:round_t. (
            is_good(N) & gv.node_has_locked_detail(N,B,R)) -> R <=
            validator(N).r_c

        invariant forall N:node_t. forall Q:cert_t. (is_good(N) &
            validator(N).qc_processed(Q)) -> gv.node_has_processed_qc(N,Q
            )
} with gv, round_t, height_t, block_t, voted_t, cert_t, timeout_t,
    timeout_cert_t, proposal_t, shim, net.spec

isolate continuity = {
invariant forall N:node_t. forall Qd:cert_t. forall Hd,Ha,Hi:height_t. (
    is_good(N) & processor.validator(N).forest_heights(Qd,Hd,Ha) & Ha <=
     Hi & Hi <= Hd) -> processor.validator(N).forest_heights(Qd,Hd,Hi)

invariant forall Hd,Hi,Ha:height_t. (Hd = Ha & Ha <= Hi & Hi <= Hd) ->
    Hi = Hd
} with gv,processor, round_t, height_t

isolate ldc_propertiesm1 = {
        relation ldc(N:node_t, Bp:block_t, Bc:block_t)

        definition ldc(N:node_t, Bp:block_t, Bc:block_t) = Bc.parent =
            block_t.hash(Bp) & (forall Rp:round_t. (Bp.round < Rp -> Bc.
            round <= Rp)) & gv.node_has_locked(N,Bc)

        relation gdc_x(B:block_t)

        definition gdc_x(B:block_t) = exists Q:quorum_t. exists R:round_t
            . ((B.round < R & forall Ri:round_t. (B.round < Ri -> R <= Ri
            )) & (forall N:node_t. (quorum_t.member(N,Q) & is_good(N)) ->
             gv.node_has_locked_detail(N,B,R)))

        property forall B:block_t. gdc_x(B) <-> gdc_properties.gdc(B)
} with gv,gdc_properties,gdc_properties2a, round_t, height_t

isolate ldc_propertiesm2 = {
        invariant forall B:block_t. forall Bp:block_t. forall N:node_t.
```

```
          ((is_good(N) & ldc_propertiesm1.ldc(N,Bp,B) & Bp ~= block_t.
          nil) -> ldc_propertiesm1.gdc_x(Bp))
} with gv,gdc_properties,gdc_properties2a, round_t, height_t,
    ldc_propertiesm1

isolate ldc_properties = {
       invariant forall N:node_t. forall B:block_t. forall R:round_t. gv
           .node_has_locked_detail(N,B,R) -> R <= processor.validator(N)
           .r_c

       private {
              invariant forall N:node_t. forall B:block_t. forall R:
                  round_t. (is_good(N) & gv.node_has_locked_detail(N,B,R
                  )) -> exists Q:cert_t. (processor.validator(N).
                  qc_processed(Q) & Q.block = B)
       }

       invariant forall N:node_t. forall B1:block_t. forall B2:block_t.
           forall R1:round_t. forall R2:round_t. (is_good(N) & gv.
           node_has_locked_detail(N,B1,R1) & gv.node_has_locked_detail(N
           ,B2,R2) & R1 < R2) -> B1.round < B2.round

       invariant forall N:node_t. forall H:height_t. (is_good(N) &
           height_t.succ(H,processor.validator(N).chain_size)) -> (gv.
           node_has_locked(N,processor.validator(N).tip_of_chain_qc.
           block) & round_t.succ(processor.validator(N).tip_of_chain_qc.
           block.round, processor.validator(N).dc_basis_block.round) &
           processor.validator(N).dc_basis_block.parent = block_t.hash(
           processor.validator(N).tip_of_chain_qc.block))

       invariant forall N:node_t. (is_good(N) & processor.validator(N).
           locked ~= cert_t.nil) -> processor.validator(N).forest_leaf(
           processor.validator(N).locked, processor.validator(N).
           locked_height)

       invariant forall N:node_t. (is_good(N) & processor.validator(N).
           dc_basis_qc ~= cert_t.nil) -> gv.node_has_locked(N,processor.
           validator(N).dc_basis_block)

       invariant forall N:node_t. (is_good(N)) -> processor.validator(N)
           .qp_when_adding_qc = processor.validator(N).tip_of_chain_qc

       invariant forall N:node_t. (is_good(N) & processor.validator(N).
           tip_of_chain_qc ~= cert_t.nil) -> processor.validator(N).
           commit_candidates(processor.validator(N).tip_of_chain_qc,
           processor.validator(N).tip_of_chain_height)

       invariant forall N:node_t. (is_good(N) & processor.validator(N).
           tip_of_chain_qc ~= cert_t.nil) -> (processor.validator(N).
```

```
            blockchain(processor.validator(N).tip_of_chain_height) =
            processor.validator(N).tip_of_chain_qc.block & processor.
            validator(N).dc_basis_block.parent = block_t.hash(processor.
            validator(N).tip_of_chain_qc.block) & round_t.succ(processor.
            validator(N).tip_of_chain_qc.block.round, processor.validator
            (N).dc_basis_block.round))

        invariant forall N:node_t. forall H:height_t. (is_good(N) &
            height_t.succ(H,processor.validator(N).chain_size)) ->
            ldc_propertiesm1.ldc(N,processor.validator(N).blockchain(H),
            processor.validator(N).dc_basis_block)
} with processor, round_t, gv, cert_t, height_t, block_t,gdc_properties,
    ldc_propertiesm1,ldc_propertiesm2

isolate ldc_properties1 = {
        invariant forall N:node_t. forall H:height_t. (is_good(N) &
            height_t.succ(H,processor.validator(N).chain_size)) ->
            ldc_propertiesm1.ldc(N,processor.validator(N).blockchain(H),
            processor.validator(N).dc_basis_block)
} with processor,gv,ldc_properties,ldc_propertiesm1,round_t,height_t

isolate ldc_properties2 = {
        invariant forall N:node_t. forall H:height_t. (is_good(N) &
            height_t.succ(H, processor.validator(N).chain_size)) ->
            ldc_propertiesm1.gdc_x(processor.validator(N).blockchain(H))

        invariant forall Bc:block_t. forall Bgdc:block_t. forall Bp:
            block_t. (gdc_supplementary_defs.node_has_recent_lock(Bc,Bgdc
            ,Bp) & Bp ~= Bgdc & Bgdc ~= block_t.nil & Bp ~= block_t.nil)
            -> Bgdc.round < Bp.round
} with processor, round_t, gv, cert_t, height_t, block_t,
    ldc_propertiesm2, ldc_properties1,ldc_properties,
    gdc_supplementary_defs

isolate ldc_properties2a = {
        invariant forall Bc:block_t. forall Bgdc:block_t. forall Bp:
            block_t. (gdc_supplementary_defs.node_has_recent_lock(Bc,Bgdc
            ,Bp) & Bp ~= Bgdc & Bgdc ~= block_t.nil) -> Bgdc.round < Bp.
            round
} with processor, round_t, gv, cert_t, height_t, block_t,
    ldc_propertiesm2, ldc_properties1,ldc_properties,
    gdc_supplementary_defs

isolate ldc_properties3 = {
        invariant forall Bc:block_t. forall Bp:block_t. forall N:node_t.
            forall Bgdc:block_t. gdc_properties10.gdc_and_lock(Bc,Bp,N,
            Bgdc) -> gdc_supplementary_defs.node_has_recent_lock(Bc,Bgdc,
            Bp)
} with processor, gv,gdc_properties11,ldc_properties,round_t,block_t,
```

```
    gdc_properties10,gdc_properties6

isolate ldc_properties4 = {
       property forall Bc:block_t. forall Bp:block_t. forall N:node_t.
           forall Bgdc:block_t. (is_good(N) & gv.node_has_locked(N,Bc) &
            Bc.parent=block_t.hash(Bp) & Bp ~= block_t.nil &
           ldc_propertiesm1.gdc_x(Bgdc) & Bgdc.round < Bc.round) ->
           gdc_properties10.gdc_and_lock(Bc,Bp,N,Bgdc)
} with gdc_properties10,ldc_propertiesm1

isolate ldc_properties5 = {
       invariant forall Bc:block_t. forall Bp:block_t. forall N:node_t.
           forall Bgdc:block_t. is_good(N) & gv.node_has_locked(N,Bc) &
           Bc.parent=block_t.hash(Bp) & Bp ~= block_t.nil &
           ldc_propertiesm1.gdc_x(Bgdc) & Bgdc.round < Bc.round & Bgdc
           ~= block_t.nil & Bp ~= Bgdc -> gdc_supplementary_defs.
           node_has_recent_lock(Bc,Bgdc,Bp) & Bp ~= Bgdc & Bgdc ~=
           block_t.nil & Bp ~= block_t.nil
} with gv,processor,ldc_properties3,ldc_properties2,ldc_properties4

isolate ldc_properties6 = {
       invariant forall Bc:block_t. forall Bp:block_t. forall N:node_t.
           forall Bgdc:block_t. is_good(N) & gv.node_has_locked(N,Bc) &
           Bc.parent=block_t.hash(Bp) & Bp ~= block_t.nil &
           ldc_propertiesm1.gdc_x(Bgdc) & Bgdc.round < Bc.round & Bgdc
           ~= block_t.nil & Bp ~= Bgdc -> Bgdc.round < Bp.round
} with gv,processor,ldc_properties2,ldc_properties5

isolate ldc_properties3a = {
       invariant forall Bc:block_t. forall Bp:block_t. forall Bgdc:
           block_t. gdc_properties10a.gdc_and_quorum(Bc,Bp,Bgdc) ->
           gdc_supplementary_defs.node_has_recent_lock(Bc,Bgdc,Bp)
} with processor, gv,gdc_properties11a,ldc_properties,round_t,block_t,
    gdc_properties10a,gdc_properties6

isolate ldc_properties4a = {
       property forall Bc:block_t. forall Bp:block_t. forall Bgdc:
           block_t. (gv.quorum_of_votes(Bc,Bp) & Bc.parent=block_t.hash(
           Bp) & Bp ~= block_t.nil & ldc_propertiesm1.gdc_x(Bgdc) & Bgdc
           .round < Bc.round) -> gdc_properties10a.gdc_and_quorum(Bc,Bp,
           Bgdc)
} with gdc_properties10a,ldc_propertiesm1

isolate ldc_properties5a = {
       invariant forall Bc:block_t. forall Bp:block_t. forall Bgdc:
           block_t. (gv.quorum_of_votes(Bc,Bp) & Bc.parent=block_t.hash(
           Bp) & Bp ~= block_t.nil & ldc_propertiesm1.gdc_x(Bgdc) & Bgdc
           .round < Bc.round & Bgdc ~= block_t.nil & Bp ~= Bgdc) -> (
           gdc_supplementary_defs.node_has_recent_lock(Bc,Bgdc,Bp) & Bp
```

```
            ~= Bgdc & Bgdc ~= block_t.nil & Bp ~= block_t.nil)
} with gv,processor,ldc_properties3a,ldc_properties2,ldc_properties4a,
    gdc_properties,round_t,block_t

isolate ldc_properties6a = {
       invariant forall Bc:block_t. forall Bp:block_t. forall Bgdc:
           block_t. is_good(N) & gv.quorum_of_votes(Bc,Bp) & Bc.parent=
           block_t.hash(Bp) & Bp ~= block_t.nil & ldc_propertiesm1.gdc_x
           (Bgdc) & Bgdc.round < Bc.round & Bgdc ~= block_t.nil & Bp ~=
           Bgdc -> Bgdc.round < Bp.round

       relation gdc_chain_condition(Bc:block_t,Bp:block_t,Bgdc:block_t)

       definition gdc_chain_condition(Bc:block_t,Bp:block_t,Bgdc:block_t
           ) = gv.quorum_of_votes(Bc,Bp) & Bc.parent=block_t.hash(Bp) &
           Bp ~= block_t.nil & ldc_propertiesm1.gdc_x(Bgdc) & Bgdc.round
            < Bc.round & Bgdc ~= block_t.nil & Bp ~= Bgdc

       invariant forall Bc:block_t. forall Bp:block_t. forall Bgdc:
           block_t. gdc_chain_condition(Bc,Bp,Bgdc) -> Bgdc.round < Bp.
           round
} with gv,processor,ldc_properties2,ldc_properties5a

isolate ldc_properties6b = {

       invariant forall Bc:block_t. forall Bp:block_t. forall Bgdc:
           block_t. (gv.quorum_of_votes(Bc,Bp) & Bc.parent=block_t.hash(
           Bp) & Bp ~= block_t.nil & gdc_properties.gdc(Bgdc) & Bgdc.
           round < Bc.round & Bgdc ~= block_t.nil & Bp ~= Bgdc) ->
           ldc_properties6a.gdc_chain_condition(Bc,Bp,Bgdc)
} with gv,processor,ldc_properties2,ldc_properties5a,ldc_properties6a,
    ldc_propertiesm1

isolate ldc_properties6c = {

       invariant forall Bc:block_t. forall Bp:block_t. forall Bgdc:
           block_t. (gv.quorum_of_votes(Bc,Bp) & Bc.parent=block_t.hash(
           Bp) & Bp ~= block_t.nil & gdc_properties.gdc(Bgdc) & Bgdc.
           round < Bc.round & Bgdc ~= block_t.nil & Bp ~= Bgdc) -> Bgdc.
           round < Bp.round

} with gv,processor,ldc_properties6a,ldc_properties6b

isolate gdc_chain_lemma = {
       relation ancestor(Bd:block_t, Ba:block_t)

       object spec = {
              after init {
                     ancestor(Bc,Bp) := true if (Bc = Bp | (Bc ~=
```

```
                        block_t.nil & Bp = block_t.nil)) else false;
            }
        }
        invariant forall Bc:block_t. forall Bp:block_t. Bc.parent =
            block_t.hash(Bp) -> Bp.round < Bc.round

        invariant forall B:block_t. block_t.nil.round <= B.round

invariant forall B1,B2:block_t. (ancestor(B1,B2) & ancestor(B2,B1)) <->
    B1 = B2

        invariant forall B1,B2,B3:block_t. (ancestor(B1,B2) & ancestor(B2
            ,B3)) -> ancestor(B1,B3)

        invariant forall B1,B2,B3:block_t. (ancestor(B1,B2) & ancestor(B1
            ,B3)) -> (ancestor(B2,B3) | ancestor(B3,B2))

        relation block_parent(Bc:block_t, Bp:block_t)

        definition block_parent(Bc:block_t, Bp:block_t) = (ancestor(Bc,Bp
            ) & Bc ~= Bp & (forall B:block_t. (ancestor(Bc,B) & Bc ~= B)
            -> ancestor(Bp,B)))

        invariant forall Bc:block_t. forall Bp:block_t. (block_parent(Bc,
            Bp) <-> Bc.parent=block_t.hash(Bp))
} with round_t,block_t,gv,quorum_propagation,processor,ldc_properties6a,
    ldc_propertiesm1

isolate gdc_chain_lemma1 = {
        relation qov_after_gdc(R:round_t,Bgdc:block_t)

        definition qov_after_gdc(R:round_t, Bgdc:block_t) = forall B:
            block_t. (B.round <= R & gdc_properties.gdc(Bgdc) & Bgdc.
            round < B.round & quorum_propagation.block_has_qov(B) & Bgdc
            ~= block_t.nil) -> (gdc_chain_lemma.ancestor(B,Bgdc))

        property forall Bgdc:block_t. forall R1,R2:round_t. (R1 <= R2 &
            qov_after_gdc(R2,Bgdc)) -> qov_after_gdc(R1,Bgdc)
} with round_t,block_t,ldc_properties6c,ldc_propertiesm1,gv,processor,
    quorum_propagation

isolate gdc_chain_lemma2 = {
        invariant forall B:block_t. forall Bp:block_t. (
            quorum_propagation.block_has_qov(B) & B.parent = block_t.hash
            (Bp)) -> gv.quorum_of_votes(B,Bp)
} with round_t,block_t,ldc_properties6c,ldc_propertiesm1,gv,processor,
    quorum_propagation

isolate gdc_chain_lemma2a = {
```

```
       invariant forall Bc:block_t. forall Bp:block_t. forall Bgdc:
           block_t. (quorum_propagation.block_has_qov(Bc) & Bc.parent=
           block_t.hash(Bp) & Bp ~= block_t.nil & gdc_properties.gdc(
           Bgdc) & Bgdc.round < Bc.round & Bgdc ~= block_t.nil & Bp ~=
           Bgdc )-> (Bgdc.round < Bp.round)
} with round_t,block_t,ldc_properties6c,gv,processor,gdc_chain_lemma2

isolate gdc_chain_lemma3 = {
       invariant forall Bc:block_t. forall Bp:block_t. forall Bgdc:
           block_t. (quorum_propagation.block_has_qov(Bc) & Bc.parent=
           block_t.hash(Bp) & Bp ~= block_t.nil & gdc_properties.gdc(
           Bgdc) & Bgdc.round < Bc.round & Bgdc ~= block_t.nil & Bp ~=
           Bgdc )-> (quorum_propagation.block_has_qov(Bp))
} with round_t,block_t,ldc_properties6c,ldc_propertiesm1,gv,processor,
    quorum_propagation,gdc_chain_lemma1

isolate gdc_chain_lemma4 = {
       invariant forall Bc:block_t. forall Bp:block_t. forall Bgdc:
           block_t. (quorum_propagation.block_has_qov(Bc) & Bc.parent=
           block_t.hash(Bp) & Bp ~= block_t.nil & gdc_properties.gdc(
           Bgdc) & Bgdc.round < Bc.round & Bgdc ~= block_t.nil & Bp ~=
           Bgdc )-> (quorum_propagation.block_has_qov(Bp) & Bgdc.round <
            Bp.round & gdc_properties.gdc(Bgdc))

       invariant forall Bp:block_t. forall Bgdc:block_t. (
           quorum_propagation.block_has_qov(Bp) & Bgdc.round < Bp.round
           & gdc_properties.gdc(Bgdc) & Bgdc ~= block_t.nil &
           gdc_chain_lemma1.qov_after_gdc(Bp.round,Bgdc)) -> (
           gdc_chain_lemma.ancestor(Bp,Bgdc))

       invariant forall Bc:block_t. forall Bp:block_t. forall Bgdc:
           block_t. (quorum_propagation.block_has_qov(Bc) & Bc.parent=
           block_t.hash(Bp) & Bp ~= block_t.nil & gdc_properties.gdc(
           Bgdc) & Bgdc.round < Bc.round & Bgdc ~= block_t.nil & Bp ~=
           Bgdc & gdc_chain_lemma1.qov_after_gdc(Bp.round,Bgdc)) -> (
           gdc_chain_lemma.ancestor(Bp,Bgdc))
} with round_t,block_t,gv,processor,quorum_propagation,gdc_chain_lemma1,
    gdc_chain_lemma2a

isolate gdc_chain_lemma5 = {
       invariant forall Bc:block_t. forall Bp:block_t. forall Bgdc:
           block_t. (Bc.parent=block_t.hash(Bp) & gdc_chain_lemma.
           ancestor(Bp,Bgdc)) -> gdc_chain_lemma.ancestor(Bc,Bgdc)

       invariant forall Bc:block_t. forall Bp:block_t. forall Bgdc:
           block_t. (quorum_propagation.block_has_qov(Bc) & Bc.parent=
           block_t.hash(Bp) & Bp ~= block_t.nil & gdc_properties.gdc(
           Bgdc) & Bgdc.round < Bc.round & Bgdc ~= block_t.nil & Bp ~=
           Bgdc & gdc_chain_lemma1.qov_after_gdc(Bp.round,Bgdc)) -> (
```

```
            gdc_chain_lemma.ancestor(Bc,Bgdc))

        invariant forall Bc:block_t. forall Bp:block_t. forall Bgdc:
            block_t. (quorum_propagation.block_has_qov(Bc) & Bc.parent=
            block_t.hash(Bp) & Bp ~= block_t.nil & gdc_properties.gdc(
            Bgdc) & Bgdc.round < Bc.round & Bgdc ~= block_t.nil & Bp =
            Bgdc & gdc_chain_lemma1.qov_after_gdc(Bp.round,Bgdc)) -> (
            gdc_chain_lemma.ancestor(Bc,Bgdc))

        invariant forall Bc:block_t. forall Bp:block_t. forall Bgdc:
            block_t. (quorum_propagation.block_has_qov(Bc) & Bc.parent=
            block_t.hash(Bp) & Bp ~= block_t.nil & gdc_properties.gdc(
            Bgdc) & Bgdc.round < Bc.round & Bgdc ~= block_t.nil &
            gdc_chain_lemma1.qov_after_gdc(Bp.round,Bgdc)) -> (
            gdc_chain_lemma.ancestor(Bc,Bgdc))

        invariant forall Bc:block_t. forall Bgdc:block_t. (
            quorum_propagation.block_has_qov(Bc) & Bc.parent ~= block_t.
            hash(block_t.nil) & gdc_properties.gdc(Bgdc) & Bgdc.round <
            Bc.round & Bgdc ~= block_t.nil & (forall R:round_t. R < Bc.
            round -> gdc_chain_lemma1.qov_after_gdc(R,Bgdc))) -> (
            gdc_chain_lemma.ancestor(Bc,Bgdc))

        invariant forall Bc:block_t. forall Bgdc:block_t. (
            quorum_propagation.block_has_qov(Bc) & gdc_properties.gdc(
            Bgdc) & Bgdc.round < Bc.round & Bgdc ~= block_t.nil & (forall
             R:round_t. R < Bc.round -> gdc_chain_lemma1.qov_after_gdc(R,
            Bgdc))) -> (gdc_chain_lemma.ancestor(Bc,Bgdc))
} with round_t,block_t,gv,processor,quorum_propagation,gdc_chain_lemma4,
    gdc_chain_lemma,gdc_properties14

isolate gdc_chain_lemma6 = {
        invariant forall Bc:block_t. forall Bgdc:block_t. forall R:
            round_t. (quorum_propagation.block_has_qov(Bc) &
            gdc_properties.gdc(Bgdc) & Bgdc.round < Bc.round & Bgdc ~=
            block_t.nil & round_t.succ(R,Bc.round) & gdc_chain_lemma1.
            qov_after_gdc(R,Bgdc)) -> (gdc_chain_lemma.ancestor(Bc,Bgdc))

        invariant forall Bgdc:block_t. forall R1,R2:round_t. (round_t.
            succ(R1,R2) & gdc_chain_lemma1.qov_after_gdc(R1,Bgdc)) ->
            gdc_chain_lemma1.qov_after_gdc(R2,Bgdc)

        invariant gdc_chain_lemma1.qov_after_gdc(0,Bgdc)
} with round_t,block_t,gv,processor,quorum_propagation,gdc_chain_lemma5,
    gdc_chain_lemma,gdc_properties14,gdc_chain_lemma1

isolate gdc_chain_lemma7 = {
        axiom [round_induction] {
               relation rel1(R:round_t, Bgdc:block_t)
```

```
            #-------------------------------------------------
            property (forall Bgdc:block_t. rel1(0,Bgdc)) & (forall
                Bgdc:block_t. forall R1,R2:round_t. (round_t.succ(R1,
                R2) & rel1(R1,Bgdc)) -> rel1(R2,Bgdc)) -> forall R:
                round_t. forall Bgdc:block_t. rel1(R,Bgdc)
        }

        property (forall Bgdc:block_t. gdc_chain_lemma1.qov_after_gdc(0,
            Bgdc)) & (forall Bgdc:block_t. forall R1,R2:round_t. (round_t
            .succ(R1,R2) & gdc_chain_lemma1.qov_after_gdc(R1,Bgdc)) ->
            gdc_chain_lemma1.qov_after_gdc(R2,Bgdc)) -> forall R:round_t.
             forall Bgdc:block_t. gdc_chain_lemma1.qov_after_gdc(R,Bgdc)
        proof {
            apply round_induction
        }

        invariant forall R:round_t. gdc_chain_lemma1.qov_after_gdc(R,Bgdc
            )

        property forall Bgdc:block_t. forall B:block_t. (gdc_properties.
            gdc(Bgdc) & Bgdc.round < B.round & quorum_propagation.
            block_has_qov(B) & Bgdc ~= block_t.nil & gdc_chain_lemma1.
            qov_after_gdc(B.round,Bgdc)) -> gdc_chain_lemma.ancestor(B,
            Bgdc)

        invariant forall Bgdc:block_t. forall B:block_t. gdc_chain_lemma1
            .qov_after_gdc(B.round,Bgdc)

        invariant forall Bgdc:block_t. forall B:block_t. (gdc_properties.
            gdc(Bgdc) & Bgdc.round < B.round & quorum_propagation.
            block_has_qov(B) & Bgdc ~= block_t.nil) -> gdc_chain_lemma.
            ancestor(B,Bgdc)
} with round_t,block_t,gv,processor,gdc_chain_lemma6,gdc_chain_lemma,
    gdc_chain_lemma1

isolate gdc_chain_lemma8 = {
        invariant forall N:node_t. forall H:height_t. (is_good(N) &
            height_t.succ(H, processor.validator(N).chain_size)) ->
            ldc_propertiesm1.gdc_x(processor.validator(N).blockchain(H))

        invariant forall N:node_t. (is_good(N) & 0 < processor.validator(
            N).tip_of_chain_height) -> height_t.succ(processor.validator(
            N).pred_of_chain_height,processor.validator(N).
            tip_of_chain_height)

        invariant forall N:node_t. forall Q:cert_t. forall H:height_t. (
            is_good(N) & processor.validator(N).locked = cert_t.nil) -> ~
            processor.validator(N).forest_store(Q,H)
```

```
        invariant forall N:node_t. (is_good(N) & 0 < processor.validator(
            N).chain_size) -> height_t.succ(processor.validator(N).
            tip_of_chain_height,processor.validator(N).chain_size)
} with height_t,block_t,processor,gv,ldc_properties2

isolate gdc_chain_lemma9 = {
        invariant forall N:node_t. (is_good(N) & 0 < processor.validator(
            N).chain_size) -> gdc_properties.gdc(processor.validator(N).
            blockchain(processor.validator(N).tip_of_chain_height))

        invariant forall N:node_t. (is_good(N) & 0 < processor.validator(
            N).tip_of_chain_height) -> (gdc_properties.gdc(processor.
            validator(N).blockchain(processor.validator(N).
            tip_of_chain_height)) & processor.validator(N).blockchain(
            processor.validator(N).tip_of_chain_height).parent = block_t.
            hash(processor.validator(N).blockchain(processor.validator(N)
            .pred_of_chain_height)))

        invariant forall N:node_t. (is_good(N) & 0 < processor.validator(
            N).chain_size) -> processor.validator(N).blockchain(0).parent
             = block_t.hash(block_t.nil)

        invariant forall N:node_t. (is_good(N) & 0 < processor.validator(
            N).chain_size) -> quorum_propagation.block_has_qov(processor.
            validator(N).blockchain(processor.validator(N).
            tip_of_chain_height))

        invariant forall N1:node_t. forall N2:node_t. (is_good(N1) &
            is_good(N2) & 0 < processor.validator(N1).chain_size & 0 <
            processor.validator(N2).chain_size) -> (gdc_properties.gdc(
            processor.validator(N1).blockchain(processor.validator(N1).
            tip_of_chain_height)) & gdc_properties.gdc(processor.
            validator(N2).blockchain(processor.validator(N2).
            tip_of_chain_height)) & quorum_propagation.block_has_qov(
            processor.validator(N1).blockchain(processor.validator(N1).
            tip_of_chain_height)) & quorum_propagation.block_has_qov(
            processor.validator(N2).blockchain(processor.validator(N2).
            tip_of_chain_height)))

        invariant forall N1:node_t. forall N2:node_t. (is_good(N1) &
            is_good(N2) & 0 < processor.validator(N1).chain_size & 0 <
            processor.validator(N2).chain_size & processor.validator(N1).
            blockchain(processor.validator(N1).tip_of_chain_height).round
             < processor.validator(N2).blockchain(processor.validator(N2)
            .tip_of_chain_height).round) -> (gdc_properties.gdc(processor
            .validator(N1).blockchain(processor.validator(N1).
            tip_of_chain_height)) & processor.validator(N1).blockchain(
            processor.validator(N1).tip_of_chain_height) ~= block_t.nil &
             quorum_propagation.block_has_qov(processor.validator(N2).
```

```
            blockchain(processor.validator(N2).tip_of_chain_height)) &
            processor.validator(N1).blockchain(processor.validator(N1).
            tip_of_chain_height).round < processor.validator(N2).
            blockchain(processor.validator(N2).tip_of_chain_height).round
            )

        invariant forall N1:node_t. forall N2:node_t. (gdc_properties.gdc
            (processor.validator(N1).blockchain(processor.validator(N1).
            tip_of_chain_height)) & processor.validator(N1).blockchain(
            processor.validator(N1).tip_of_chain_height) ~= block_t.nil &
             quorum_propagation.block_has_qov(processor.validator(N2).
            blockchain(processor.validator(N2).tip_of_chain_height)) &
            processor.validator(N1).blockchain(processor.validator(N1).
            tip_of_chain_height).round < processor.validator(N2).
            blockchain(processor.validator(N2).tip_of_chain_height).round
            ) -> gdc_chain_lemma.ancestor(processor.validator(N2).
            blockchain(processor.validator(N2).tip_of_chain_height),
            processor.validator(N1).blockchain(processor.validator(N1).
            tip_of_chain_height))

        invariant forall N1:node_t. forall N2:node_t. (gdc_properties.gdc
            (processor.validator(N2).blockchain(processor.validator(N2).
            tip_of_chain_height)) & processor.validator(N2).blockchain(
            processor.validator(N2).tip_of_chain_height) ~= block_t.nil &
             quorum_propagation.block_has_qov(processor.validator(N1).
            blockchain(processor.validator(N1).tip_of_chain_height))&
            processor.validator(N2).blockchain(processor.validator(N2).
            tip_of_chain_height).round < processor.validator(N1).
            blockchain(processor.validator(N1).tip_of_chain_height).round
            ) -> gdc_chain_lemma.ancestor(processor.validator(N1).
            blockchain(processor.validator(N1).tip_of_chain_height),
            processor.validator(N2).blockchain(processor.validator(N2).
            tip_of_chain_height))
} with height_t,block_t,processor,gv,gdc_chain_lemma8,ldc_propertiesm1,
    gdc_properties16,gdc_chain_lemma,gdc_chain_lemma7,gdc_properties

isolate gdc_chain_lemma10 = {
        invariant forall N1:node_t. forall N2:node_t. (is_good(N1) &
            is_good(N2) & 0 < processor.validator(N1).chain_size & 0 <
            processor.validator(N2).chain_size & processor.validator(N2).
            blockchain(processor.validator(N2).tip_of_chain_height).round
             = processor.validator(N1).blockchain(processor.validator(N1)
            .tip_of_chain_height).round) -> (processor.validator(N1).
            blockchain(processor.validator(N1).tip_of_chain_height) =
            processor.validator(N2).blockchain(processor.validator(N2).
            tip_of_chain_height))

        invariant forall N1:node_t. forall N2:node_t. (is_good(N1) &
            is_good(N2) & 0 < processor.validator(N1).chain_size & 0 <
```

```
            processor.validator(N2).chain_size) -> (gdc_chain_lemma.
            ancestor(processor.validator(N1).blockchain(processor.
            validator(N1).tip_of_chain_height),processor.validator(N2).
            blockchain(processor.validator(N2).tip_of_chain_height)) |
            gdc_chain_lemma.ancestor(processor.validator(N2).blockchain(
            processor.validator(N2).tip_of_chain_height),processor.
            validator(N1).blockchain(processor.validator(N1).
            tip_of_chain_height)))
} with gv, height_t, processor, ldc_properties, round_t, block_t,
    gdc_chain_lemma9,gdc_chain_lemma

isolate ancestor_not_sibling = {
        relation orphan(B:block_t)

        definition orphan(B:block_t) = forall Bp:block_t. B.parent ~=
            block_t.hash(Bp)

        invariant (forall B1:block_t. forall B2:block_t. (gdc_chain_lemma
            .ancestor(B1,B2) & B1.parent = B2.parent) -> B1 = B2)
} with gdc_chain_lemma,round_t,gdc_properties,block_t,processor,gv


isolate blockchain_ancestor = {
        axiom [height_induction] {
                relation rel1(H:height_t)
                #------------------------------------------------
                property (rel1(0) & (forall H1,H2:height_t. (height_t.succ
                    (H1,H2) & rel1(H1)) -> rel1(H2))) -> forall H:height_t
                    . rel1(H)
        }

        relation blockchain_height_ancestor(H:height_t)

        definition blockchain_height_ancestor(H:height_t) = forall N:
            node_t. forall Hl:height_t. (is_good(N) & processor.validator
            (N).blockchain(H) ~= block_t.nil & Hl <= H) ->
            gdc_chain_lemma.ancestor(processor.validator(N).blockchain(H)
            ,processor.validator(N).blockchain(Hl))

        invariant forall N:node_t. (is_good(N) & processor.validator(N).
            blockchain(H) ~= block_t.nil) -> gdc_chain_lemma.ancestor(
            processor.validator(N).blockchain(0),processor.validator(N).
            blockchain(0))

        invariant blockchain_height_ancestor(0)

        invariant forall N:node_t. forall H1,H2:height_t. (is_good(N) &
            processor.validator(N).blockchain(H2) ~= block_t.nil &
            height_t.succ(H1,H2)) -> processor.validator(N).blockchain(H2
```

```
            ).parent = block_t.hash(processor.validator(N).blockchain(H1)
            )

        invariant forall N:node_t. forall H1,H2:height_t. (is_good(N) &
            processor.validator(N).blockchain(H2) ~= block_t.nil &
            height_t.succ(H1,H2)) -> gdc_chain_lemma.block_parent(
            processor.validator(N).blockchain(H2),processor.validator(N).
            blockchain(H1))

        invariant forall H1,H2:height_t. (height_t.succ(H1,H2) &
            blockchain_height_ancestor(H1)) -> blockchain_height_ancestor
            (H2)

        property (blockchain_height_ancestor(0) & (forall H1,H2:height_t.
             (height_t.succ(H1,H2) & blockchain_height_ancestor(H1)) ->
            blockchain_height_ancestor(H2))) -> forall H:height_t.
            blockchain_height_ancestor(H)
        proof {
                apply height_induction
        }
} with gv,processor,gdc_chain_lemma,height_t

isolate blockchain_ancestor1 = {
        invariant forall H:height_t. blockchain_ancestor.
            blockchain_height_ancestor(H)
} with gv,processor,gdc_chain_lemma,height_t,blockchain_ancestor

isolate blockchain_safety_lemma0 = {
        invariant forall N:node_t. forall H:height_t. (is_good(N) & H <=
            processor.validator(N).tip_of_chain_height & processor.
            validator(N).blockchain(H) ~= block_t.nil) -> 0 < processor.
            validator(N).chain_size

        invariant forall N:node_t. forall H:height_t. (is_good(N) & H <=
            processor.validator(N).tip_of_chain_height & processor.
            validator(N).blockchain(H) ~= block_t.nil) -> processor.
            validator(N).blockchain(processor.validator(N).
            tip_of_chain_height) ~= block_t.nil

        invariant forall N:node_t. forall H:height_t. (is_good(N) & H <=
            processor.validator(N).tip_of_chain_height & processor.
            validator(N).blockchain(H) ~= block_t.nil) -> gdc_chain_lemma
            .ancestor(processor.validator(N).blockchain(processor.
            validator(N).tip_of_chain_height),processor.validator(N).
            blockchain(H))
} with blockchain_ancestor,blockchain_ancestor1, gv,processor,block_t,
    round_t,height_t,gdc_chain_lemma8

isolate blockchain_safety_lemma1 = {
```

```
invariant forall N:node_t. (is_good(N) & 0 < processor.validator(N).
    chain_size) -> height_t.succ(processor.validator(N).
    tip_of_chain_height,processor.validator(N).chain_size)

       invariant forall N:node_t. forall H:height_t. (is_good(N) &
           processor.validator(N).tip_of_chain_height < H & 0 <
           processor.validator(N).chain_size) -> processor.validator(N).
           blockchain(H) = block_t.nil

       invariant forall N:node_t. forall H:height_t. (is_good(N) &
           processor.validator(N).blockchain(H) ~= block_t.nil) -> (0 <
           processor.validator(N).chain_size)

       invariant forall N1:node_t. forall N2:node_t. forall H:height_t.
           (is_good(N1) & is_good(N2) & processor.validator(N1).
           blockchain(H) ~= block_t.nil & processor.validator(N2).
           blockchain(H) ~= block_t.nil) -> (gdc_chain_lemma.ancestor(
           processor.validator(N1).blockchain(processor.validator(N1).
           tip_of_chain_height),processor.validator(N2).blockchain(
           processor.validator(N2).tip_of_chain_height)) |
           gdc_chain_lemma.ancestor(processor.validator(N2).blockchain(
           processor.validator(N2).tip_of_chain_height),processor.
           validator(N1).blockchain(processor.validator(N1).
           tip_of_chain_height)))

       invariant forall N1:node_t. forall N2:node_t. forall H:height_t.
           (is_good(N1) & is_good(N2) & processor.validator(N1).
           blockchain(H) ~= block_t.nil & processor.validator(N2).
           blockchain(H) ~= block_t.nil) -> (gdc_chain_lemma.ancestor(
           processor.validator(N1).blockchain(processor.validator(N1).
           tip_of_chain_height),processor.validator(N1).blockchain(H)) &
            gdc_chain_lemma.ancestor(processor.validator(N2).blockchain(
           processor.validator(N2).tip_of_chain_height),processor.
           validator(N2).blockchain(H)))

       invariant forall Ba1,Ba2,B1,B2:block_t. (gdc_chain_lemma.ancestor
           (B1,Ba1) & gdc_chain_lemma.ancestor(B2,Ba2) & (
           gdc_chain_lemma.ancestor(B1,B2) | gdc_chain_lemma.ancestor(B2
           ,B1))) -> (gdc_chain_lemma.ancestor(Ba1,Ba2) |
           gdc_chain_lemma.ancestor(Ba2,Ba1))

       invariant forall N1:node_t. forall N2:node_t. forall H:height_t.
           (is_good(N1) & is_good(N2) & processor.validator(N1).
           blockchain(H) ~= block_t.nil & processor.validator(N2).
           blockchain(H) ~= block_t.nil) -> (gdc_chain_lemma.ancestor(
           processor.validator(N1).blockchain(H),processor.validator(N2)
           .blockchain(H)) | gdc_chain_lemma.ancestor(processor.
           validator(N2).blockchain(H),processor.validator(N1).
           blockchain(H)))
```

```
} with gv,processor,gdc_chain_lemma8,height_t,gdc_chain_lemma10,
    blockchain_safety_lemma0,gdc_chain_lemma

isolate blockchain_safety_lemma2 = {
       relation blockchain_safe(H:height_t)

       definition blockchain_safe(H:height_t) = forall N1:node_t. forall
            N2:node_t. (is_good(N1) & is_good(N2) & processor.validator(
           N1).blockchain(H) ~= block_t.nil & processor.validator(N2).
           blockchain(H) ~= block_t.nil) -> (processor.validator(N1).
           blockchain(H) = processor.validator(N2).blockchain(H))

       invariant forall N:node_t. (is_good(N) & processor.validator(N).
           blockchain(0) ~= block_t.nil) -> processor.validator(N).
           blockchain(0).parent = block_t.hash(block_t.nil)

       invariant forall N1,N2:node_t. (is_good(N1) & is_good(N2) &
           processor.validator(N1).blockchain(0) ~= block_t.nil &
           processor.validator(N2).blockchain(0) ~= block_t.nil) -> (
           gdc_chain_lemma.ancestor(processor.validator(N1).blockchain
           (0),processor.validator(N2).blockchain(0)) | gdc_chain_lemma.
           ancestor(processor.validator(N2).blockchain(0),processor.
           validator(N1).blockchain(0)))

       invariant forall N1,N2:node_t. (is_good(N1) & is_good(N2) &
           processor.validator(N1).blockchain(0) ~= block_t.nil &
           processor.validator(N2).blockchain(0) ~= block_t.nil &
           blockchain_safe(H1)) -> processor.validator(N1).blockchain(0)
           .parent = processor.validator(N2).blockchain(0).parent

       invariant blockchain_safe(0)

       property forall H1,H2:height_t. height_t.succ(H1,H2) -> 0 < H2

       invariant forall H1,H2:height_t. forall N1,N2:node_t. (height_t.
           succ(H1,H2) & is_good(N1) & is_good(N2) & processor.validator
           (N1).blockchain(H2) ~= block_t.nil & processor.validator(N2).
           blockchain(H2) ~= block_t.nil) -> (gdc_chain_lemma.ancestor(
           processor.validator(N1).blockchain(H2),processor.validator(N2
           ).blockchain(H2)) | gdc_chain_lemma.ancestor(processor.
           validator(N2).blockchain(H2),processor.validator(N1).
           blockchain(H2)))

       invariant forall H1,H2:height_t. forall N1,N2:node_t. (height_t.
           succ(H1,H2) & is_good(N1) & is_good(N2) & processor.validator
           (N1).blockchain(H2) ~= block_t.nil & processor.validator(N2).
           blockchain(H2) ~= block_t.nil & blockchain_safe(H1)) ->
           processor.validator(N1).blockchain(H2).parent = processor.
           validator(N2).blockchain(H2).parent
```

```
        invariant forall H1,H2:height_t. forall N1,N2:node_t. (height_t.
            succ(H1,H2) & is_good(N1) & is_good(N2) & processor.validator
            (N1).blockchain(H2) ~= block_t.nil & processor.validator(N2).
            blockchain(H2) ~= block_t.nil & blockchain_safe(H1)) ->
            processor.validator(N1).blockchain(H2) = processor.validator(
            N2).blockchain(H2)

        invariant forall H1,H2:height_t. (height_t.succ(H1,H2) &
            blockchain_safe(H1)) -> blockchain_safe(H2)

        property (blockchain_safe(0) & (forall H1,H2:height_t. (height_t.
            succ(H1,H2) & blockchain_safe(H1)) -> blockchain_safe(H2)))
            -> forall H:height_t. blockchain_safe(H)
        proof {
                apply blockchain_ancestor.height_induction
        }

        invariant forall H:height_t. blockchain_safe(H)
} with height_t, block_t, gv, processor,ancestor_not_sibling,
    blockchain_safety_lemma1

isolate global_properties = {
        invariant forall N:node_t. forall H:height_t. (is_good(N) &
            processor.validator(N).blockchain(H) ~= block_t.nil) ->
            exists Q:cert_t. (processor.validator(N).qc_processed(Q) & Q.
            block = processor.validator(N).blockchain(H))

        invariant [round_safety_statement] forall N1:node_t. forall N2:
            node_t. forall H1:height_t. forall H2:height_t. (is_good(N1)
            & is_good(N2) & processor.validator(N1).blockchain(H1) ~=
            block_t.nil & processor.validator(N2).blockchain(H2) ~=
            block_t.nil & processor.validator(N1).blockchain(H1).round =
            processor.validator(N2).blockchain(H2).round ) -> processor.
            validator(N1).blockchain(H1) = processor.validator(N2).
            blockchain(H2)

        invariant forall N:node_t. forall B:block_t. (is_good(N) & gv.
            node_has_voted(N,B)) -> B.parent = block_t.hash(block_t.nil)
            | exists Bp:block_t. (B.parent = block_t.hash(Bp) & gv.
            node_has_locked(N,Bp))

        invariant forall N1:node_t. forall N2:node_t. forall H:height_t.
            (is_good(N1) & is_good(N2) & processor.validator(N1).
            blockchain(H) ~= block_t.nil & processor.validator(N2).
            blockchain(H) ~= block_t.nil) -> processor.validator(N1).
            blockchain(H) = processor.validator(N2).blockchain(H)
} with gv, height_t, processor, ldc_properties, round_t, block_t,
    blockchain_safety_lemma2
```

```
} with gv, round_t, height_t, block_t, cert_t, shim, net.spec, gdc_properties,
    gdc_properties2,gdc_properties2a
```

B.7. `check_all.sh`

```
date
ivy_check trace=true complete=fo isolate=basic_safety classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.ancestor_not_sibling classic_safety.
    ivy
date
ivy_check trace=true macro_finder=false isolate=basic_safety.
    blockchain_ancestor classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.blockchain_ancestor1 classic_safety.
    ivy
date
ivy_check trace=true isolate=basic_safety.blockchain_safety_lemma0
    classic_safety.ivy
date
ivy_check trace=true macro_finder=false isolate=basic_safety.
    blockchain_safety_lemma1 classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.blockchain_safety_lemma2
    classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.continuity classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.gdc_chain_lemma classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.gdc_chain_lemma1 classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.gdc_chain_lemma10 classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.gdc_chain_lemma2 classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.gdc_chain_lemma2a classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.gdc_chain_lemma3 classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.gdc_chain_lemma4 classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.gdc_chain_lemma5 classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.gdc_chain_lemma6 classic_safety.ivy
date
ivy_check trace=true macro_finder=false isolate=basic_safety.gdc_chain_lemma7
    classic_safety.ivy
```

```
date
ivy_check trace=true isolate=basic_safety.gdc_chain_lemma8 classic_safety.ivy
date
ivy_check trace=true macro_finder=false isolate=basic_safety.gdc_chain_lemma9
    classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.global_properties classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.ldc_properties classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.ldc_properties1 classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.ldc_properties2 classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.ldc_properties2a classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.ldc_properties3 classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.ldc_properties3a classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.ldc_properties4 classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.ldc_properties4a classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.ldc_properties5 classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.ldc_properties5a classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.ldc_properties6 classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.ldc_properties6a classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.ldc_properties6b classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.ldc_properties6c classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.ldc_propertiesm1 classic_safety.ivy
date
ivy_check trace=true macro_finder=false isolate=basic_safety.ldc_propertiesm2
    classic_safety.ivy
date
ivy_check trace=true isolate=basic_safety.processor classic_safety.ivy
date
ivy_check trace=true isolate=block_t classic_safety.ivy
date
ivy_check trace=true isolate=cert_t classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties classic_safety.ivy
date
```

```
ivy_check trace=true isolate=gdc_properties1 classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties10 classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties10a classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties11 classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties11a classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties12 classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties13 classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties14 classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties15 classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties16 classic_safety.ivy
date
ivy_check trace=true macro_finder=false isolate=gdc_properties2 classic_safety.
    ivy
date
ivy_check trace=true macro_finder=false isolate=gdc_properties2a classic_safety
    .ivy
date
ivy_check trace=true isolate=gdc_properties3 classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties4 classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties5 classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties6 classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties7 classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties7m1 classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties7m2 classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties8 classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties9 classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties9a classic_safety.ivy
date
ivy_check trace=true isolate=gdc_properties9am1 classic_safety.ivy
date
ivy_check trace=true isolate=gdc_supplementary_defs classic_safety.ivy
```

```
date
ivy_check trace=true isolate=gv classic_safety.ivy
date
ivy_check trace=true isolate=hash_t classic_safety.ivy
date
ivy_check trace=true isolate=height_t.iso classic_safety.ivy
date
ivy_check trace=true isolate=index.iso classic_safety.ivy
date
ivy_check trace=true isolate=node_t.iso_iter classic_safety.ivy
date
ivy_check trace=true isolate=node_t.iter.iso classic_safety.ivy
date
ivy_check trace=true isolate=proposal_t classic_safety.ivy
date
ivy_check trace=true isolate=quorum_propagation classic_safety.ivy
date
ivy_check trace=true isolate=quorum_propagationm1 classic_safety.ivy
date
ivy_check trace=true isolate=quorum_propagationm2 classic_safety.ivy
date
ivy_check trace=true isolate=quorum_t classic_safety.ivy
date
ivy_check trace=true isolate=round_t.iso classic_safety.ivy
date
ivy_check trace=true isolate=shim classic_safety.ivy
date
ivy_check trace=true complete=fo isolate=this classic_safety.ivy
date
ivy_check trace=true isolate=timeout_cert_t classic_safety.ivy
date
ivy_check trace=true isolate=timeout_t classic_safety.ivy
date
ivy_check trace=true isolate=voted_t classic_safety.ivy
date
```