An Executable K Model of Ethereum 2.0 Beacon Chain Phase 0 Specification

DENIS BOGDANAS, Runtime Verification Inc.
DAEJUN PARK, Runtime Verification Inc.
CHRIS HATHHORN, Runtime Verification Inc.
MUSAB A. ALTURKI, Runtime Verification Inc.
GRIGORE ROŞU, Runtime Verification, Inc. and University of Illinois at Urbana-Champaign

The beacon chain is the main proof-of-stake block chain in Ethereum 2.0 maintaining information about validators, attestations, crosslinks, and other components of the protocol. The structure and operation of the beacon chain is defined by its reference implementation in Python developed by the Ethereum Foundation. This report describes an executable formal model in the K framework of Ethereum's Beacon Chain Phase 0 specification. We highlight the structure of the model, explain how the beacon chain state transition and the supporting functions are specified and then outline how the model is validated using the standard beacon chain test suite. We present general test coverage analysis and a summary of our findings and recommendations. The full specification is available online at https://github.com/runtimeverification/beacon-chain-spec.

1 INTRODUCTION

Consensus in the Ethereum network has so far been based on a proof-of-work protocol. However, due to computational efficiency and energy consumption considerations, Ethereum is undergoing a transition into a proof-of-stake-based consensus protocol, namely Casper [1], in which the choice of the next block-proposing validator node is proportional to its share at stake in the network (rather than its hash power). Casper's design aims at guaranteeing desirable safety and liveness properties of transaction blocks. Furthermore, to significantly increase transaction throughput and address scalability concerns, Sharding [7] has also been proposed as another major update to Ethereum. A shard is a partition of a larger blockchain system that maintains its own state and transaction history. Both Casper and Sharding, along with several other planned major updates, will be part of a major future version of Ethereum, called Ethereum 2.0 (a.k.a. Serenity) [3]. To facilitate a smooth transition, development is planned to take place in seven phases. The first phase, Phase 0 [4], targets fully specifying the main PoS chain in Ethereum 2.0, called the beacon chain, which maintains validator records and manages validator attestations, in addition to laying the foundation for a specification of shards.

As our ultimate goal is to have a formal framework for specifying, executing and verifying Casper and Sharding, we introduce in this report a first-step towards achieving this goal, which is an executable, formal specification of the full beacon chain specification given by "Ethereum 2.0 Phase 0" [4] in the K framework. The specification is formal, defining node states as configuration patterns and specifying protocol operations and behaviors as transitions over patterns, modeled as rewrite rules. Furthermore, the specification is executable by pattern rewriting in the K tool, which provides immediately (at no extra cost) an execution engine for the protocol.

Executable specifications not only enable running simulations and animating systems, which can be a tremendously useful tool for prototyping and debugging different designs during the development process, but can also be used as reference implementations for model-based test generation and for validating other implementations. Perhaps more importantly, the specification itself can be used (through K's recently developed LLVM backend) to deploy a correct-by-construction implementation that is reasonably efficiently executable. Furthermore, such an

Authors' addresses: Denis Bogdanas, Runtime Verification Inc. denis.bogdanas@runtimeverification.com; Daejun Park, Runtime Verification Inc. daejun.park@runtimeverification.com; Chris Hathhorn, Runtime Verification Inc. chris.hathhorn@runtimeverification.com; Musab A. Alturki, Runtime Verification Inc. musab.alturki@runtimeverification.com; Grigore Roşu, Runtime Verification, Inc. University of Illinois at Urbana-Champaign, grosu@illinois.edu.

approach will seamlessly enable further refining the protocol's formal design without having to worry about modifying the implementation. Finally, the executable specification of the protocol in K is immediately available to K's (ever-expanding) arsenal of reachability, model checking and theorem proving tools, facilitating various forms of formal analysis of properties.

In addition, we aim for this Phase 0 specification to lay the foundation for both: (1) further future extensions corresponding to new features or components that may appear in later phases of development of Ethereum 2.0, and (2) further formal modeling and verification developments.

The full specification developed in this work is available online at https://github.com/runtimeverification/beacon-chain-spec. The repository includes documented specifications, compilation instructions and scripts for running the tests described in this report.

The rest of the report is organized as follows. In Section 2 below, we give an overview of the beacon chain. This is followed in Section 3 by a description of our model of the beacon chain in K. In Section 4 we outline how the model is validated against existing tests. Section 5 describes test coverage analysis, along with findings and recommendations. The report concludes in Section 6 with a discussion of possible future developments.

2 ETHEREUM 2.0 BEACON CHAIN PHASE 0

The beacon chain is the main chain of the proof-of-stake protocol layer of the upcoming Ethereum 2.0 [3]. The chain consists of a sequence of beacon blocks containing information pertaining to various aspects of the protocol, including validators and deposits, attestations and committees, justification and finalization, rewards and penalties, shards and crosslinks, and hashing and merkle-ization, in addition to the state of the underlying proof-of-work Ethereum 1.0 blockchain. The reference implementation of the beacon chain protocol that maintains this chain is given in Python [4].

The protocol progresses temporally with respect to a logical timeline in which time is divided into epochs, which are further divided into a fixed number of individual short periods of time called slots. Figure 1 (borrowed from [11]) illustrates this timeline.

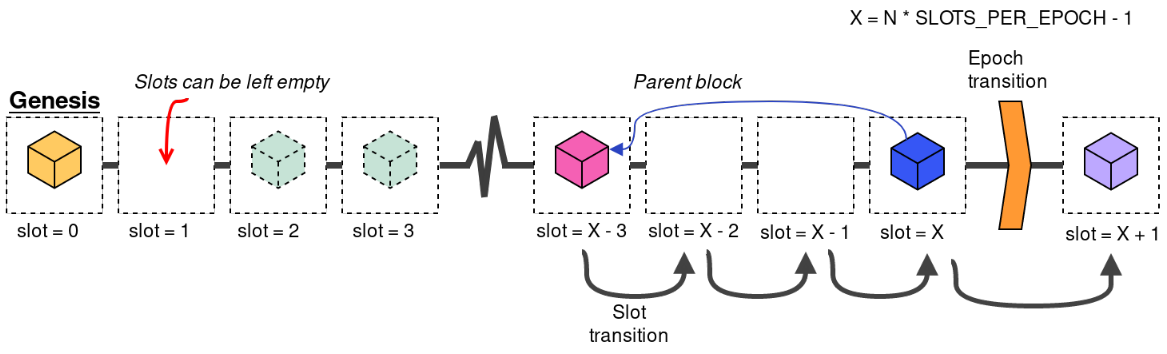

Fig. 1. An illustration of Eth 2.0 timeline [11]

During each slot, a designated validator (called a proposer) from a committee selected through a RANDAO-like shuffling process proposes a new beacon chain block, and all other validators in that committee submit their attestations (for this and possibly other blocks floating around). An attestation is a vote for a block. Attestations serve several different purposes in the protocol and are at the heart of its operation. For example, an attestation confirms progression to the block's slot, identifies the block's parent in the chain (fork choice rule), contributes to deciding finality of blocks (Casper FFG [1]) and links to the block's shard in case the block does not belong to the main chain (crosslinks).

An epoch transition takes place when progressing across epoch-boundary slots. Much of the protocol processing happens during an epoch transition. Examples include: processing justification and finalization of blocks according to Casper, updating crosslinks, rewarding well-behaving validators and penalizing ones that deviated from the protocol, and processing deposits to join in as validators and requests to withdraw deposits and exit from the system.

Two core components of the specification of the beacon chain are: (1) the beacon chain state and (2) the beacon chain state transition function. The beacon chain state maintains information about the chain itself (where it currently fits within a view of the network), the current time slot, a full registry of validators and their states, all pending attestations, crosslinks to shards, finality information, a link to the underlying Ethereum 1.0 blockchain, among other things. The state includes all pieces of information needed for the proper functioning of the protocol.

The state is processed by the second core component of the specification, which is the beacon chain state transition function. Given the next beacon block to be processed, the function transforms a given beacon chain state (pre-state) into a new state (post-state) that reflects the results of progressing time up to the block's slot value (and performing epoch-transition processing if appropriate) and then processing the block's contents. An excerpt of the relevant part of the function's implementation in Python is shown below:

```
def state_transition(state: BeaconState, block: BeaconBlock ...) — >BeaconState:
  # Process slots (including those with no blocks) since block
  process_slots(state, block.slot)
  # Process block
  process_block(state, block)
  ...
  # Return post-state
  return state
```

Processing begins at the genesis state (the state having the genesis beacon block already processed). When processing a (non-genesis) block proposed for slot N ($N \geq 0$), and assuming the block is valid (i.e. the block passes all the internal checks specified by the state transition function and all its sub-procedures), the state transition function advances in steps its slot value to N, while initiating epoch-transition processing if an epoch boundary is crossed in the process, and then processes the block at slot N. The resulting state is the post-state of processing that block, which is also the pre-state of the next block to be processed. We note that the state transition function is defined only for valid blocks, and the validity of an entire chain beginning at the genesis block is derived recursively from the successive application of the state transition function on the sequence of blocks as they appear in the chain. More details can be found here [5].

3 MODELING THE BEACON CHAIN IN K

As stated above, the immediate objective is to have a formal specification that enables: (1) simulating the beacon state transition function. (2) running existing tests from the beacon chain test-suites, and (3) analyzing test coverage and potentially developing new tests to improve coverage. To achieve these goals, the formalization has to be executable, to animate states and analyze executions. Furthermore, the formalization needs to be done at a low-enough level of abstraction that is compatible with how tests are specified (so that the need to instrument the tests or the specification is minimized or even eliminated). Despite being fairly low-level, the formalization abstracts over signature verification since that is outside the scope of the project.

We, therefore, present an executable formalization in the K framework of the beacon chain (Phase 0) specification [4, 5], including its two core components: the beacon chain state and the beacon chain state transition function. The model is characterized by being: (1) executable by pattern rewriting in the K tool, so that an interpreter for the beacon chain state transition function is obtained directly from the specification (at no extra cost), and (2) concrete, in that its specification corresponds directly to the Python implementation of the system (modulo some specific abstractions, such as signature verification). These two characteristics not only enable simulation and testing, but also facilitate further future developments, including deriving efficiently executable correct-by-construction reference implementations and automatically generating concrete tests (see our previous work on generating reference implementations based on K specifications [9, 10]).

Furthermore, we chose to define the model using K's declarative programming features directly, and not through an intermediate representation or through some definition of Python's semantics. Although an intermediate language would likely have resulted in a more high-level specification, this approach gets us finer-grained tooling support needed in this project, such as parsing, type-checking, semantic debugging and test-coverage analysis facilities, that would otherwise be less effective.

The full specification of the model in K is available at https://github.com/runtimeverification/beacon-chain-spec. We list below the main components/files of the model:

- The core K model specification:
  - `constants-minimal.k`: definitions of system-wide constants for testing purposes, corresponding to the "minimal" test suite of beacon chain
  - `types.k`: type definitions and basic operations corresponding to Python data types and builtin functions and beacon-chain-specific types and operations needed by the model.
  - `config.k`: specification of a K configuration defining the structure and components of the beacon state
  - `hash-tree.k`: specification of the hash-tree (merkle-tree) computation functions on beacon-chain data structures
  - `beacon-chain.k`: specification of the beacon chain state transition function and all its supporting sub-functions
- Testing scripts:
  - `buildConfig.py` and `runTest.py`: scripts that load beacon chain implementation-independent tests into K

We focus in this section on the core K model. Testing and model validation is described in Section 4.

3.1 Types and Basic Operations

The beacon chain specification defines various data types on top of different Python's built-in data types. These types need to be specified in the K model (at some suitable level of abstraction). Type definitions in K are generally specified by introducing non-terminals (syntactic categories in a context-free grammar) in BNF notation with appropriate production rules. Most of the basic types are modeled using the three basic types built into K, namely Bool for Boolean, Int for integers, String for strings, and K's list and map constructors. For example, a hash value is modeled by a fixed-length string (with the fixed length being enforced by the semantics):

```
syntax Hash ::= String
```

And the zeroed hash value is defined a string of zeros of length 32 (in hexadecimal notation matching the one used in the original implementation):

```
syntax Hash ::= defaultHash() [function]
rule defaultHash()  => {defaultBytes32()}:>String
```

Furthermore, certain other types are modeled by extended versions of K's internal types. For example, although standard validator indices are modeled by (non-negative) integer values, the case of a "null" index is modeled by an the special terminal symbol `.ValidatorIndex`, extending the original integer type into the new type `ValidatorIndex`:

```
syntax ValidatorIndex ::= Int | ".ValidatorIndex"
```

In addition to the basic data types, the beacon chain implementation defines several more complex container types consisting of several fields. For each container data type in the implementation, we introduce in the K model a new type (non-terminal) that is constructed syntactically by an operator that takes the the container's fields as arguments. For example, the validator container in the implementation is declared with eight fields as follows:

```
class Validator(Container):
    pubkey: BLSPubkey
    withdrawal_credentials: Hash
    effective_balance: Gwei
    slashed: boolean
    activation_eligibility_epoch: Epoch
    activation_epoch: Epoch
    exit_epoch: Epoch
    withdrawable_epoch: Epoch
```

The corresponding type in the model is defined by the following syntactic declaration:

```
syntax Validator ::= #Validator( pubkey: BLSPubkey,
                                  withdrawalCredentials: Hash,
                                  effectiveBalance: Int,
                                  slashed: Bool,
                                  activationEligibilityEpoch: Int,
                                  activationEpoch: Int,
                                  exitEpoch: Int,
                                  withdrawableEpoch: Int
                                ) [klabel(#Validator), symbol]
```

The rule attributes klabel and symbol instruct the parser to give the unique name #Validator to nodes constructed with this rule in the abstract syntax tree, so they may be referred to easily when analyzing a state.

In addition to the type declaration itself, we introduce syntax that allows accessing fields in a container by name. For example, for the type Validator above, a validator's effective balance can be accessed using the following syntax:

```
syntax Int ::= Validator "." "effectiveBalance" [function]
rule #Validator(_,_, EffBal, _,_,_,_,_).effectiveBalance  =>EffBal
```

Finally, we also define various basic operations on types that will be useful for defining other parts of the model, including type conversion functions (e.g. converting from an integer into a sequence of bytes) and list operations that are not directly available in K (such as integer list sorting and list appending operations).

3.2 The Beacon Chain Configuration

The beacon chain state's implementation is given as a Python container class that is composed of a number of fields, including for instance, the current slot value, a link to the underlying Ethereum 1.0 blockchain, a full registry of validators and their current states, collected attestations, crosslinks to shards, and justification and finalization information. Below is an excerpt of the container declaration in Python:

```
class BeaconState(Container):
  ...
  slot: Slot
  eth1_data: Eth1Data
  validators: List[Validator, VALIDATOR_REGISTRY_LIMIT]
  balances: List[Gwei, VALIDATOR_REGISTRY_LIMIT]
  slashings: Vector[Gwei, EPOCHS_PER_SLASHINGS_VECTOR]
  current_crosslinks: Vector[Crosslink, SHARD_COUNT]
  finalized_checkpoint: Checkpoint
  ...
```

We model the state structure as a configuration in K, where each field is modeled by an appropriate cell in the configuration. For example, the configuration declaration and the part that corresponds to the excerpt above is show next:

```
configuration ⟨beacon-chain⟩
                ⟨k⟩ $PGM:KItem ⟨/k⟩
                ⟨state⟩                        //Type BeaconState in Python spec.
                  ...
                  ⟨slot⟩ 0 ⟨/slot⟩
                  ⟨eth1-data⟩ .Eth1Data ⟨/eth1-data⟩
                  ⟨validators⟩ .Map ⟨/validators⟩
                  ⟨balances⟩ .Map ⟨/balances⟩
                  ⟨slashings⟩ .Map ⟨/slashings⟩
                  ⟨current-crosslinks⟩ .Map ⟨/current-crosslinks⟩
                  ⟨finalized-checkpoint⟩ .Checkpoint ⟨/finalized-checkpoint⟩
                  ...
                ⟨/state⟩
                ⟨zerohashes-cache⟩ .Map ⟨/zerohashes-cache⟩
              ⟨/beacon-chain⟩
```

We note that in many cases, indexed lists and vectors in Python are modeled in the configuration by maps, of which the dot construct `.Map` represents the empty map. Indexed maps enable matching against specific map element patterns, such as matching against a slashed validator structure mapped to by a given index value. Obviously, the cells will be initialized with more meaningful values when the configuration is used to model a concrete state of the beacon chain (when testing for example).

Having the state modeled explicitly as a K configuration makes the definition of the state transition function more natural, as the state transition system induced by the function becomes explicit. Furthermore, the configuration may include other supporting fields, including the special computation cell k to specify the operations to be executed and the `<zerohashes-cache>` cell used as part of hash tree root computation (both shown above).

3.3 Hash-tree Computation

Several components of the beacon chain state store hashes of values and structures computed through a custom-defined hash-tree (merkle-ization) process defined in Python [6] and described more generally in this document [8]. This process is captured in the Python implementation by the `hath_tree_root` function, which computes for every value and container a single root in a hash tree structure. For example, the beacon chain state stores in its `BeaconBlockHeader` container the hash of the latest block seen and the hash of the block's parent, as per this function.

Since one main objective of the model is to use it to run existing tests directly, and since many of the these tests involve checking validity of supplied hashes, we specify the hashing function `hath_tree_root` and other supporting operations in the K model. Very generally, the corresponding function in K is defined inductively over the structure of values (basic data types, lists of values and more complex container types), and matches very closely the Python definitions.

3.4 The State Transition Function

The state transition function of the beacon chain defines the core of the beacon chain protocol behavior. The function, implemented in Python as the function `state_transition` shown above in Section 2, maps a beacon chain pre-state S_i and a beacon chain block B_i into a beacon chain post-state S_{i+1}. In a sequence of state updates (consecutive function applications) , the resulting post-state S_{i+1} is itself the pre-state of the next function application step on some block B_{i+1}, and so on. We note that the transition function is deterministic, i.e. the post-state is completely determined by the pre-state and the block. Another important observation is that the function is implemented as a non-pure function (a function with side effects) in Python that modifies the state object passed to it as an argument.

In the K model, the state transition function is specified by an operator in K that transforms the beacon chain configuration (representing the pre-state) into another beacon chain configuration configuration (representing the post-state) given a beacon block term as an argument to the operator:

```
syntax KItem ::= "state_transition" "(" block: BeaconBlock ")"
                 [klabel(state_transition), symbol]
```

As described in Section 2, there are two main consecutive steps involved: (1) advancing slots all the way to the block's slot value `process_slots`, and then (2) processing the block `process_block`. Sequencing of commands in K is naturally specified as stacking a computation on top of a continuation, using the operator ~>. For example, the state transition function is defined using the following computation stack:

```
rule state_transition(BLOCK) => process_slots(BLOCK.slot) ~> process_block(BLOCK)
```

Only when `process_slots` terminates successfully will the next computation defined by `process_block` take place, which captures the intended semantics.

3.4.1 Sequencing. This sequencing pattern is seen throughout the Python implementation and, in many cases, it is handled in the same way using user-defined computational structures in K and the stacking operator ~>. For example, advancing a slot across an epoch boundary invokes the following function:

```
def process_epoch(state: BeaconState) −>None:
    process_justification_and_finalization(state)
    process_crosslinks(state)
    process_rewards_and_penalties(state)
    process_registry_updates(state)
    process_slashings(state)
    process_final_updates(state)
```

which consists of a sequence of function calls on the beacon chain state. This sequence is modeled by the following definition in K:

```
syntax KItem ::= "process_epoch" "(" ")"
rule ⟨k⟩ process_epoch() =>
               process_justification_and_finalization()
            ~> process_crosslinks()
            ~> process_rewards_and_penalties()
            ~> process_registry_updates()
            ~> process_slashings()
            ~> process_final_updates()
     ... ⟨/k⟩
```

Notice that the corresponding operators in K have arity 0, as they operate on the current configuration containing the computation cell k to which they belong.

3.4.2 Variables and assignments. Another prevalent pattern in the Python implementation, which is prevalent in any implementation written in an imperative programming language, is the use of local variables and temporary values. Since K is declarative, temporary variables and values need to be modeled. This is handled mainly in one of two ways (depending on the complexity of the code being translated): (1) lambda applications in K using the `#fun` construct, and (2) auxiliary function declarations taking extra arguments.

The expression `#fun(X => E) E'` corresponds to evaluating the argument `E'` first to a value V, and then returning the result of evaluating `E` with every (free) occurrence of X in E replaced by V (call-by-value semantics of lambda). An example of using this construct to bind a value to a local name in the K model is shown below:

```
syntax IndexedAttestation ::= "get_indexed_attestation" "(" Attestation ")" [function]
rule get_indexed_attestation(#Attestation(AggregationBits, DATA, CustodyBits, SIG)) =>
    #fun(AttestingIndices =>
    #fun(CustodyBit1Indices =>
    #fun(CustodyBit0Indices =>#IndexedAttestation(
                               sortIntList(CustodyBit0Indices),
                               sortIntList(CustodyBit1Indices),
                               DATA,
                               SIG
                            )
    )(listDiff(AttestingIndices, CustodyBit1Indices))
    )(get_attesting_indices(DATA, CustodyBits))
    )(get_attesting_indices(DATA, AggregationBits))
    requires listDiff(get_attesting_indices(DATA, CustodyBits),
                      get_attesting_indices(DATA, AggregationBits)) ==K .IntList
```

The rule constructs a new indexed attestation structure populated by values bound to local names introduced by three nested instances of `#fun`. For example, the name `AttestingIndices` is bound to the value of the expression `get_attesting_indices(DATA, AggregationBits)`. Note also that the rule is conditional with the condition specified using the `requires` clause.

In other, less complex statements, declaring an additional operator that takes extra arguments corresponding to temporary local values is more appropriate. `process_slot` (for processing a single slot) is an example:

```
syntax KItem ::= "process_slot" "(" ")"
rule ⟨k⟩ process_slot()
```

```
        => processSlotAux(hash_tree_root_state(), SLOT %Int SLOTS_PER_HISTORICAL_ROOT)
    ...⟨/k⟩
    ⟨slot⟩ SLOT ⟨/slot⟩
```

where the hash-tree root of the beacon chain state and the current slot value are bound to local names in the newly introduced auxiliary function `processSlotAux`. Another example is the function for computing the list of active validator indices:

```
syntax IntList ::= "get_active_validator_indices" "(" epoch: Int ")" [function]
rule [[ get_active_validator_indices(EP) =>
        getActiveValidatorIndicesAux(.IntList, 0, VALIDATORS, EP) ]]
    ⟨validators⟩ VALIDATORS ⟨/validators⟩
```

The auxiliary function `getActiveValidatorIndicesAux` adds three new arguments for computing the index list, which are local values for that function call. Note that the use of the double square brackets `[[` and `]]` is necessary to allow this function-defining rule to access contents of other cells in the state.

3.4.3 Assertions. Assertions implement checks that must pass for the computation of the state transition function to proceed. If at any point an assertion fails, as a result of an unexpected state, computation halts with an error. This type of behavior is modeled in the K specification by reaching a stuck configuration, a non-terminal configuration to which no rewrite applies (non-terminal in that the computation cell <k> is not the empty computation `.K`). The way to achieve this is to add rules guarded by Boolean expressions corresponding to assertions wherever they appear in the implementation. For instance, the function `get_block_root_at_slot` begins with an assertion:

```
def get_block_root_at_slot(state: BeaconState, slot: Slot) − >Hash:
    assert slot ⟨state.slot ⟨= slot + SLOTS_PER_HISTORICAL_ROOT
    return state.block_roots[slot % SLOTS_PER_HISTORICAL_ROOT]
```

The function returns the block root at the given slot value only when the assertion holds. Otherwise, it halts with a failure. This is modeled by the following conditional rule:

```
syntax Hash ::= "get_block_root_at_slot" "(" Int ")" [function]
rule [[ get_block_root_at_slot(SLOT)
        => {BLOCKROOTS[ SLOT %Int SLOTS_PER_HISTORICAL_ROOT ]}:⟩Hash ]]
    ⟨slot⟩ StateSLOT ⟨/slot⟩
    ⟨block-roots⟩ BLOCKROOTS ⟨/block-roots⟩
  requires SLOT ⟨Int StateSLOT andBool StateSLOT ⟨=Int (SLOT +IntSLOTS_PER_HISTORICAL_ROOT)
```

Since this rule is the only rule that defines the operator `get_block_root_at_slot` in the specification, execution stops while evaluating this operation and a stuck configuration is reached when the condition fails. On the other hand, if the condition holds, execution proceeds normally.

3.4.4 Control flow – conditionals. Conditional branching in the Python implementation is specified in the K model using either: (1) K's internal conditional expressions `#if ... #fi`, or (1) conditional rules with K's `requires` clause. When used with K's localized rewriting, the use of the internal conditional expression to capture branching behaviors can result in compact yet quite readable definitions. An example is shown below:

```
syntax KItem ::= "decrease_balance" "(" ValidatorIndex "," Int ")"
rule ⟨k⟩ decrease_balance(ValIndex, Delta) =>. ...⟨/k⟩
     ⟨balances⟩...
```

```
      ValIndex |-> (BAL  => #if Delta >Int BAL #then 0 #else BAL -Int Delta #fi)
    ...</balances>
```

The rule defines how the balance of a validator is decreased in the beacon chain state. The balance rewrites to the reduced value if it is non-negative, and to 0 otherwise.

In other more complex branching statements (larger conditions and/or branches), the use of conditional rules is generally more appropriate and can result in a more readable specification. Conditional rules also have the advantage of exposing branching to rule-based coverage analysis tools in K, but that usually comes at the expense of a less concise specification (see Section 4). For example, the operation `initiate_validator_exit` corresponding to the identically named function in Python for initiating the process of exiting the system of a given validator has the following definition:

```
syntax KItem ::= "initiate_validator_exit" "(" Int ")"

rule initiate_validator_exit(INDEX) =>.K
    requires getValidator(INDEX).exitEpoch =/=K FAR_FUTURE_EPOCH

rule initiate_validator_exit(INDEX) =>
        initiateValidatorExitAux(INDEX, exitQueueEpochAux(INDEX))
    requires getValidator(INDEX).exitEpoch ==K FAR_FUTURE_EPOCH
```

There are two cases: (1) either the validator has already exited, which is handled by the first rule above, in which case execution terminates (`.K` is the empty computation), or (2) the validator has not exited the system, handled by the second rule, in which case the rest of the computations involved in initiating the exit process is delegated to another auxiliary operator `initiateValidatorExitAux`.

3.4.5 Control flow – looping. Since K is declarative, the specification has to encode iterative behaviors encountered in Python definitions, including simple counter-based loops, looping over elements of a list structure, general Boolean-condition-guarded loops and implicit looping in list/set comprehension notation. This process involves defining auxiliary, recursively defined functions with conditional rules capturing the loop and exit branches of an iterative construct. This encoding is generally quite verbose (especially when compared with Python's comprehension notation), but is inescapable without defining an intermediate looping construct in K. The advantage, however, is that the encoding readily enables a more detailed coverage analysis (see Section 4).

A simple example of translating looping constructs is is `process_slots` function definition:

```
def process_slots(state: BeaconState, slot: Slot) − >None:
    assert state.slot <= slot
    while state.slot <slot:
        process_slot(state)
        if (state.slot + 1) % SLOTS_PER_EPOCH == 0:
            process_epoch(state)
        state.slot += Slot(1)
```

The function increments the state slot value in steps, invoking epoch processing operations when crossing epoch boundaries, up to the slot value of the block (given as an argument). The operator declaration in the K model that corresponds to this function is:

```
syntax KItem ::= "process_slots" "(" Int ")" [klabel(process_slots), symbol]
```

We note here again that the state does not need to be supplied as an argument. The configuration in which computations take place encode the state. The first step in the encoding of this function is to check that the assertion holds, before proceeding to the loop encoding, given by processSlotsLoop (whose declaration is not shown):

```
rule ⟨k⟩ process_slots(SLOT)  =>processSlotsLoop(SLOT) ⋯⟨/k⟩
      ⟨slot⟩ StateSLOT ⟨/slot⟩
    requires StateSLOT ⟨=Int SLOT
```

If the assertion fails, i.e. the condition of the rule above does not hold, the rule does not fire, and since there is no other rule that rewrites process_slots, rewriting gets stuck at this term signaling a dynamic (runtime) error. Otherwise, we proceed with processing the loop. The looping branch of the while statement has the following encoding:

```
rule ⟨k⟩ processSlotsLoop(SLOT)  =>
              process_slot()
           ~> #if (StateSLOT +Int1) %Int SLOTS_PER_EPOCH ==K 0 #then process_epoch()
              #else .K #fi
           ~> incrementSlot()
           ~> processSlotsLoop(SLOT)
       ⋯⟨/k⟩
       ⟨slot⟩ StateSLOT ⟨/slot⟩
    requires StateSLOT ⟨Int SLOT
```

The body of the loop consists of three steps: process one slot, invoke epoch processing if needed, and then increment the state slot. The steps are done in sequence, modeled stacking computational structures using ~>, one structure per step. the fourth and last computational structure in the sequence introduced by the rule is processSlotsLoop itself, which recursively causes the loop to repeat, as required. Note that this rule applies only when loop's Boolean condition holds.

The exit branch of the loop is modeled by the following rule:

```
rule ⟨k⟩ processSlotsLoop(SLOT)  =>.K ... ⟨/k⟩
      ⟨slot⟩ StateSLOT ⟨/slot⟩
    requires StateSLOT ⟩=Int SLOT
```

which simply terminates execution, when the loop's Boolean condition fails.

More complex looping structures, e.g. iterating over elements of a list container type or list/set comprehension expressions, require more elaborate encodings to keep track of indices and stopping conditions, and typically require defining additional operators to model internal filter and map operations. Examples include compute_committee and get_winning_crosslink_and_attesting_indices among others.

4 VALIDATING THE MODEL

The Ethereum Foundation provides a rich test suite for the beacon chain protocol [2]. It exists in 2 formats. First is a set of unit tests-style tests specifically for the reference implementation of beacon chain, written in Python. It tests both the state_transition function and separate steps within the state transition, like process_attestation or process_slot. To ensure that various production versions of beacon chain clients conform to the reference implementation, they provide same tests in a second, implementation-independent format. It essentially contains the BeaconState object before the tested operation, after the tested operation, and operation parameters. For example for `process_attestation` the test will contain 3 objects: pre-state `BeaconChain`, post-state `BeaconChain`, and the `Attestation` object. If the test is for a failure case, e.g. it tests that the client rejects invalid input data, then there's no post-state. All objects are given in yaml, which is a version of xml. These conformance tests are generated from Python unit tests, and thus the two test formats represent the same testing conditions.

We tested our K beacon chain specification using the implementation-independent test format. More precisely, K beacon chain runs all tests designed for Beacon Chain Phase 0 [2], except a small number of tests that are specifically testing BLS signature validation. BLS validation is outside the scope of the present project.

5 EVALUATING TEST COVERAGE

The beacon chain test suite was designed with code coverage in mind. When running the Python-style test suite over the reference implementation, code coverage shows only one line not covered by tests, on the execution path of `process_rewards_and_penalties`. This is the only large function that doesn't have its own test suite, something the beacon chain developers are aware of and working on it.

The K Framework also has a test coverage tool which we can use. Its granularity is at the rule level. For example, it detects for each rule whether it was applied or not during test suite execution. At first, we were skeptical whether the K coverage tool could expose any non-covered functionality, given the attention to coverage that beacon chain developers have. However, it turned out to be quite useful and provided a number of surprising results.

We initially expected K coverage to be more coarse-grained than Python, and thus less informative. This was mainly because most `if` statements are modeled through K's `#if_#then_#else_#fi` construct, and do not represent separate rules. Thus, K's coverage would not be able to distinguish between the then and else cases. However, K beacon chain is more fine-grained than the Python's implementation in other aspects, as we show below. Next we enumerate useful findings resulting from K's code coverage analysis tool.

First, we identified a line of code that is only covered by tests with `bls_setting` enabled (while not necessarily related o BLS verification). This has been reported to the beacon chain developers and they confirmed that any non-BLS functionality must be covered by non-BLS tests, and thus this has been indeed recognized as an issue.

Second, there is a while loop of the form `while True: ...` in `get_beacon_proposer_index`, which had to be modeled through three separate rules in the K model. As a consequence of this, coverage discovered that in all tests this loop does only one iteration. Normally coverage tools do not distinguish the numbers of iterations loops take, but in this particular case, it would be useful to have tests that exhibit behaviors taking multiple iterations of the loop, at least to demonstrate that it is possible.

Furthermore, and most interestingly, we identified twelve other rules not covered by tests, in addition to what Python's coverage tool reports, on valid execution paths. All of these rules relate to Python's set/list comprehension statements, which is perhaps the most elegant language feature specific to Python. A list comprehension construct is very semantically rich involving quite a lot of logic, including enumerating elements of a list, filtering them and constructing a new list out of the results. Yet it is treated as just one line of code by Python coverage tool. As a result, as long as the comprehension statement is reached, Python considers it completely covered, without distinguishing various scenarios of what happens inside. For example, if list comprehension always returns an empty list, Python will consider it covered.

In K, a list comprehension with filtering usually has to be modeled through 3 separate rules. For example. `get_matching_target_attestations` has the following Python definition:

```
def get_matching_target_attestations(state: BeaconState, epoch: Epoch)
    − > Sequence[PendingAttestation]:
  return [
    a for a in get_matching_source_attestations(state, epoch)
      if a.data.target.root == get_block_root(state, epoch)
  ]
```

while in the K model, this function is specified by the following syntactic structures and rules:

```
syntax PendingAttestationList ::=
                        "get_matching_target_attestations" "(" Int ")" [function]
rule get_matching_target_attestations(EPOCH) =>
  filterOutNonMatchingTargets(get_matching_source_attestations(EPOCH),
                              EPOCH,
                              .PendingAttestationList)

syntax PendingAttestationList ::= filterOutNonMatchingTargets(
                                      PendingAttestationList,
                                      Int,
                                      PendingAttestationList) [function]
rule filterOutNonMatchingTargets(PA PAList =>PAList, EPOCH, RES =>RES +append PA)
  requires PA.data.target.root ==K get_block_root(EPOCH)
rule filterOutNonMatchingTargets(PA PAList =>PAList, EPOCH, RES)
  requires PA.data.target.root =/=K get_block_root(EPOCH)
rule filterOutNonMatchingTargets(.PendingAttestationList, EPOCH, RES) =>RES
```

As a consequence of this, K's coverage distinguishes three cases: (a) list comprehension performs filtering, (b) list comprehension does not perform filtering but returns a non-empty result, (c) list comprehension returns an empty result.

For example, for the code above, the second rule for `filterOutNonMatchingTargets` is not covered, meaning that filtering is never performed. All other eleven detections for loop coverage are on execution paths of `process_rewards_and_penalties`. New tests for this function are being developed by the beacon chain developers at the time of this writing and we plan to re-run coverage when they are added to the test suite, to see if there are still missing cases remaining. For the other two detections described above, we contributed tests that cover them to beacon chain project.

We also plan to implement an ad-hoc context sensitive code coverage for `#if` construct, to eliminate granularity issue we described above. Initial experiments show that this leads to more functionality not being covered.

6 CONCLUSION

We presented an executable formal model in K of Ethereum 2.0 Phase 0 Specification of the beacon chain. The model is distinguished by being fairly comprehensive (models the entirety of the beacon chain state transition function - modulo signature verification), executable (yielding an interpreter for the protocol), and concrete (specified at a low-enough level of abstraction that enables running the standard tests unmodified). We have validated the model against the beacon chain specification test suite, analyzed test coverage using K's rule-based test coverage tool, identified a number of cases for which test coverage can be improved and proposed some additional tests.

Moving forward, there are several interesting directions for further development and research. The model can be extended to support Phase 1 of the beacon chain specification on sharding, once that specification becomes ready for such development. Similar testing and code coverage analysis can be performed there as well. Another direction is to formally specify and verify desirable properties about the beacon chain transition function, which can be either low-level invariants expressible at this level of abstraction or high-level properties (such as safety and liveness properties) that can be expressed and verified at some higher-level abstraction that the concrete model is shown to suitably refine.

ACKNOWLEDGMENTS

We thank Danny Ryan, Carl Beekhuizen and Martin Lundfall from the Ethereum Foundation for their contribution and comments. This work was funded by the Ethereum Foundation.

REFERENCES

[1] Vitalik Buterin and Virgil Griffith. 2017. Casper the Friendly Finality Gadget. *CoRR* abs/1710.09437 (2017). arXiv:1710.09437 http://arxiv.org/abs/1710.09437

[2] Ethereum Foundation. 2019. Eth 2.0 spec tests. https://github.com/ethereum/eth2.0-spec-tests

[3] Ethereum Foundation. 2019. Ethereum 2.0. https://github.com/ethereum/eth2.0-specs

[4] Ethereum Foundation. 2019. Ethereum 2.0 Phase 0 – The Beacon Chain. https://github.com/ethereum/eth2.0-specs/blob/dev/specs/core/0_beacon-chain.md

[5] Ethereum Foundation. 2019. Phase 0 accompanying resource. https://notes.ethereum.org/@djrtwo/ByHlx-j6V?type=view

[6] Ethereum Foundation. 2019. Python implementation of the Simple Serialization encoding and decoding. https://github.com/ethereum/py-ssz

[7] Ethereum Foundation. 2019. Sharding FAQ. https://github.com/ethereum/wiki/wiki/Sharding-FAQ

[8] Ethereum Foundation. 2019. SimpleSerialize (SSZ). https://github.com/ethereum/eth2.0-specs/blob/dev/specs/simple-serialize.md

[9] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. 2018. KEVM: A Complete Semantics of the Ethereum Virtual Machine. In *2018 IEEE 31st Computer Security Foundations Symposium.* IEEE, 204–217.

[10] Theodoros Kasampalis, Dwight Guth, Brandon Moore, Traian Florin Serbanuta, Yi Zhang, Daniele Filaretti, Virgil Serbanuta, Ralph Johnson, and Grigore Rosu. 2019. IELE: A Rigorously Designed Language and Tool Ecosystem for the Blockchain. In *Proceedings of the 23 International Symposium on Formal Methods (FM'19)*.

[11] Diederik Loerakker. 2019. ETH 2.0 educational resources. https://github.com/protolambda/eth2-docs/blob/master/README.md#ssz-encoding