Week 6 Development Track - EL & CL Specs

Guest speaker

- Hsiao-Wei Wang, Ethereum Foundation Consensus R&D team
- Sam Wilson, Ethereum, EIP editor, wallets?

Summary notes

- Edited by Chloe Zhu
- Online version: https://ab9jvcjkej.feishu.cn/docx/U9W5dKqIxopyNBxbdf4cXB1InPc

Consensus Layer Specs

CL specs repo: https://github.com/ethereum/consensus-specs

3 purposes of the CL specs repo

- It's a collection of Ethereum core consensus specifications
- It's executable and verifable
- It's test vector generator

Flow of adding new feature patch

- Implement new features in Pyspec markdown files
- Release new Pyspec with test vector suite
    - Important in the CL R&D process as this process finds basic bugs before implementation
    - Test repo: https://github.com/ethereum/consensus-spec-tests
- CL clients implement and test against test vectors

Why use Python?

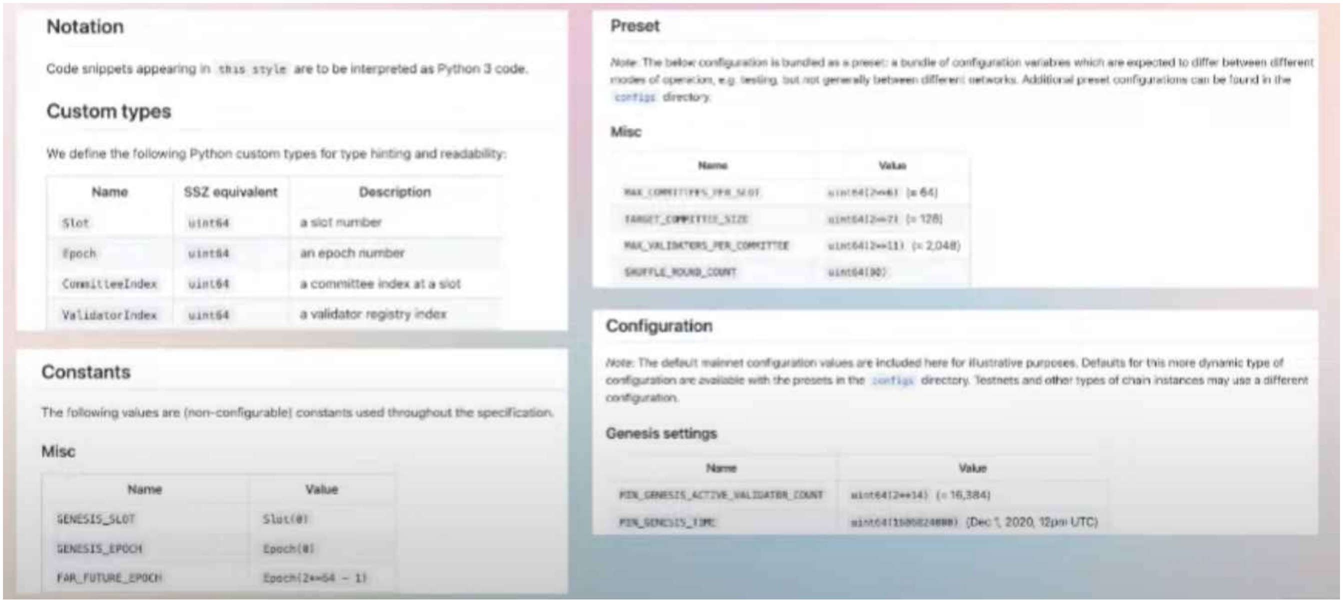
- Cause it's very readable to devs
- Main principle of the CL specs: Readability & Simplification

How to read the specs?

- Folder structure

- Specs folder: incl. all the CL hardforks already happened on the mainnet & WIP research projects
  - Each hardfork folder usually incl. several mark down files, beacon-chain.md would be a good start for each hardfork specs
  - Features folder: WIP CL-related research projects
- SSZ folder: SSZ (Simple Serialize) containers
  - Info on SSZ: https://ethereum.org/en/developers/docs/data-structures-and-encoding/ssz/
  - Devs also use SSZ hash tree root as the digests of consensus objects

- How to read the specs
  - Type and Values definitions: Values are usually defined in the markdown tables, incl. custom types, constants, preset, configuration



Notation

Code snippets appearing in `this style` are to be interpreted as Python 3 code.

Custom types

We define the following Python custom types for type hinting and readability:

Name	SSZ equivalent	Description
`Slot`	`uint64`	a slot number
`Epoch`	`uint64`	an epoch number
`CommitteeIndex`	`uint64`	a committee index at a slot
`ValidatorIndex`	`uint64`	a validator registry index

Constants

The following values are (non-configurable) constants used throughout the specification.

Misc

Name	Value
`GENESIS_SLOT`	`Slot(0)`
`GENESIS_EPOCH`	`Epoch(0)`
`FAR_FUTURE_EPOCH`	`Epoch(2**64 - 1)`

Preset

Note: The below configuration is bundled as a preset: a bundle of configuration variables which are expected to differ between different modes of operation, e.g. testing, but not generally between different networks. Additional preset configurations can be found in the `configs` directory.

Misc

Name	Value
`MAX_COMMITTEES_PER_SLOT`	`uint64(2**6)` (= 64)
`TARGET_COMMITTEE_SIZE`	`uint64(2**7)` (= 128)
`MAX_VALIDATORS_PER_COMMITTEE`	`uint64(2**11)` (= 2,048)
`SHUFFLE_ROUND_COUNT`	`uint64(90)`

Configuration

Note: The default mainnet configuration values are included here for illustrative purposes. Defaults for this more dynamic type of configuration are available with the presets in the `configs` directory. Testnets and other types of chain instances may use a different configuration.

Genesis settings

Name	Value
`MIN_GENESIS_ACTIVE_VALIDATOR_COUNT`	`uint64(2**14)` (= 16,384)
`MIN_GENESIS_TIME`	`uint64(1606824000)` (Dec 1, 2020, 12pm UTC)

  - State transition function
    - Python function can also be written in code block to describe the consensus rule
    - E.g. the assertion in phase 0/ beacon-chain.md's state transition function

Beacon chain state transition function

The post-state corresponding to a pre-state `state` and a signed block `signed_block` is defined as `state_transition(state, signed_block)`. State transitions that trigger an unhandled exception (e.g. a failed `assert` or an out-of-range list access) are considered invalid. State transitions that cause a `uint64` overflow or underflow are also considered invalid.

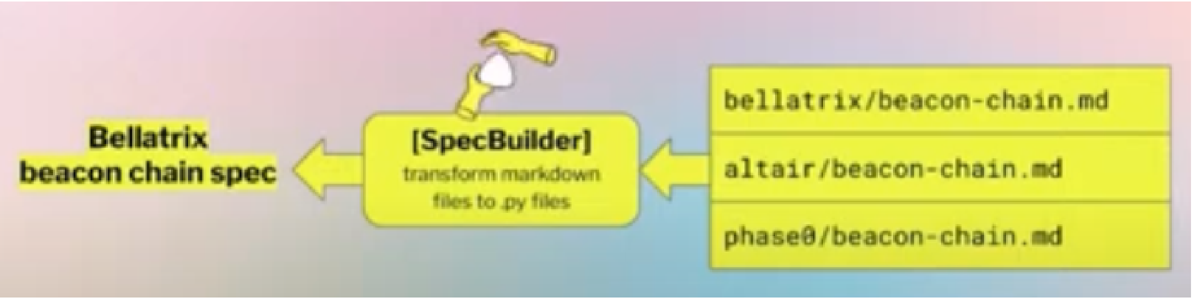
```python
def state_transition(state: BeaconState, signed_block: SignedBeaconBlock,
    block = signed_block.message
    # Process slots (including those with no blocks) since block
    process_slots(state, block.slot)
    # Verify signature
    if validate_result:
        assert verify_block_signature(state, signed_block)
    # Process block
    process_block(state, block)
    # Verify state root
    if validate_result:
        assert block.state_root == hash_tree_root(state)
```

Useful resources to understand CL

- Vitalik's annotated spec: https://github.com/ethereum/annotated-spec
- Ben Edgington's Upgrading Ethereum Book: https://eth2book.info/capella/

The elf in setup.py

- Convert the mark down file into Python, and extend the previous hard forks



Demo: How to use Pyspec?

- Installation (Python 3.8+)
    - Download the source code:

```
git clone https://github.com/ethereum/consensus-specs.git
```

    - Install with Makefile commands

```
cd consensus-specs
make install test && make pyspec
```

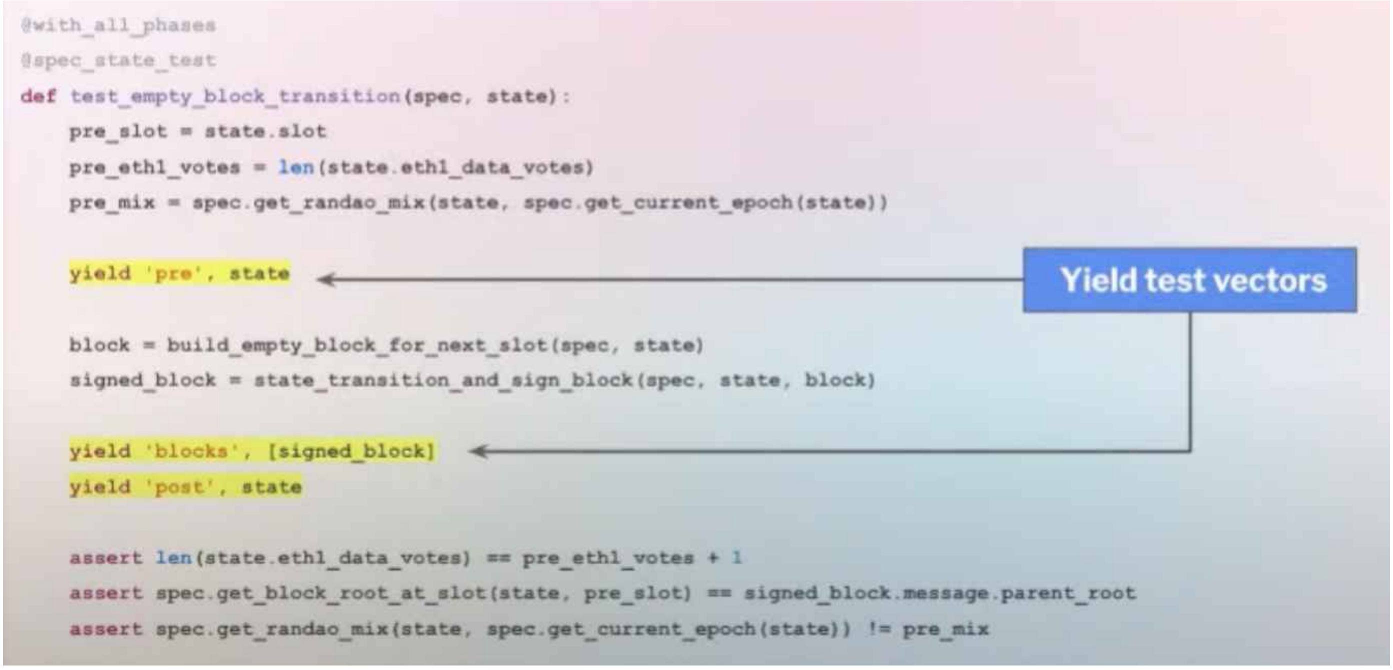
- Run your first pyspec program

```
>> from eth2spec.bellatrix import mainnet as spec
>> hello = b"Hello World"
>> body = spec.BeaconBlockBody(
>>     graffiti=hello + b'\0' * (32 - len(hello))
>> )
>> block = spec.BeaconBlock(body=body)
>> print(block.body.graffiti.decode("utf-8"))
Hello World
```

- Write your first pyspec test case

```
@with_all_phases
@spec_state_test
def test_empty_block_transition(spec, state):
    pre_slot = state.slot
    pre_eth1_votes = len(state.eth1_data_votes)
    pre_mix = spec.get_randao_mix(state, spec.get_current_epoch(state))

    yield 'pre', state

    block = build_empty_block_for_next_slot(spec, state)
    signed_block = state_transition_and_sign_block(spec, state, block)

    yield 'blocks', [signed_block]
    yield 'post', state

    assert len(state.eth1_data_votes) == pre_eth1_votes + 1
    assert spec.get_block_root_at_slot(state, pre_slot) == signed_block.message.parent_root
    assert spec.get_randao_mix(state, spec.get_current_epoch(state)) != pre_mix
```

- Pyspec as the test vector generator



- Documents
  - Pyspec: https://github.com/ethereum/consensus-specs/blob/dev/tests/README.md
  - Test formats: https://github.com/ethereum/consensus-specs/blob/dev/tests/formats/README.md

How to contribute to Pyspec?

- Level 1: Look through the specs files to learn about the specs logic & help review it
- Level 2: Help refactor the codebase
- Level 3: Try to hack some new edge test cases
- Level 4: Submit to bug bounty (https://ethereum.org/en/bug-bounty/)

Q&A

- How CL specs interact with the EL side? Eg. the state transition function
  - In the fork-choice.md file, the **on_block** function defines whether a block is received. If the assertions are satisfied, then it will call the **state_transition** function from beacon-chain.md
  - Reference file
    - Fork-choice.md: https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/fork-choice.md
    - Beacon-chain.md - State transition function: https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/beacon-chain.md#beacon-chain-state-transition-function
- Any testing for chain reorgs?

- In fork-choice.md, abstract **dataclass** is defined and used to handle reorg and state transition.

  ```
  @dataclass
  class Store(object):
      time: uint64
      genesis_time: uint64
      justified_checkpoint: Checkpoint
      finalized_checkpoint: Checkpoint
      unrealized_justified_checkpoint: Checkpoint
      unrealized_finalized_checkpoint: Checkpoint
      proposer_boost_root: Root
      equivocating_indices: Set[ValidatorIndex]
      blocks: Dict[Root, BeaconBlock] = field(default_factory=dict)
      block_states: Dict[Root, BeaconState] = field(default_factory=dict)
      block_timeliness: Dict[Root, boolean] = field(default_factory=dict)
      checkpoint_states: Dict[Checkpoint, BeaconState] = field(default_factory=dict)
      latest_messages: Dict[ValidatorIndex, LatestMessage] = field(default_factory=dict)
      unrealized_justifications: Dict[Root, Checkpoint] = field(default_factory=dict)
  ```

  - Justified checkpoint state is used as the basic stable state
    - **update_checkpoints** function is used to update the justified & finalized checkpoint
    - **get_head** function is used to execute LMD-GHOST fork choice
  - In some cases, reorg might happen and **Proposer head and reorg helpers** are gonna help to define the situation and get proposer head
    - Proposer head and reorg helpers
      - `is_head_late`
      - `is_shuffling_stable`
      - `is_ffg_competitive`
      - `is_finalization_ok`
      - `is_proposing_on_time`
      - `is_head_weak`
      - `is_parent_strong`
      - `get_proposer_head`
- Reference file: https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/fork-choice.md

- The biggest challenge when developing CL specs?
  - Performance is always a shared challenge. Another one is the conversion between mark down file and python program.

Ethereum Execution Layer Specs (EELS)

EELS repo: https://github.com/ethereum/execution-specs

Tldr of the EELS

- From the **Yellow paper** (academic paper with dense math notation) to now **fully executable python implementation of the Ethereum EL**

What's EELS

- It's Python reference implementation of most of a Ethereum client
  - What EELS doesn't do: networking, fork choice, reorgs
  - Other than the above, EELS pretty much builds the entire EL
- Team members
  - Guruprasad Kamath: github.com/gurukamath
  - Peter Davies: https://github.com/petertdavies
  - And contributors from all around the world

The Yellow paper and its main problems & benefits

History of the Yellow paper

- Created around 2014 by Gavin Wood
- Creative Commons Attribution Share-Alike (CC-BY-BA) Version 4.0 license, which is a more restrictive one
  - i.e. If you use it, you have to release your stuff under the same license
- The Yellow paper defines the blockchain, fork choice, state, transaction, block, and the rest (incl. Gas, contracts, virtual machine, RLP, modified MPT, precompiles, and EVM instructions)

Main problems

- The math notions in the paper are succinct and difficult to understand
- Audience
  - The Yellow Paper is inaccessible to most programmers, while programmers are the ones who need to understand it the most
  - Core EIP/ EIP authors rarely use this notation
- Untestable
  - The Yellow paper is (mostly) human language
  - No way to test the spec itself or use the spec to fill tests

Key benefits

- Succinct
- Formal

- Algorithm independent

EELS and its main benefits & problems

History of EELS

- Created around May 2021 by Consensys' Quilt team
- Most recently maintained by the EELS team at the Ethereum Foundation
- Creative Commons Zero v1.0 Universal license

Why EELS?

- To escape the frustration of trying to understand the Yellow paper
- No "snapshot" of the current state of Ethereum
- More accessible to programmers with Python
- The same document that specifies Ethereum also fills the automated tests for it

EELS walkthrough

- Directary structure: Tour forks
    - Each fork has its folder and within each folder has the complete copy of Ethereum specs
        - Link: https://github.com/ethereum/execution-specs/tree/master/src/ethereum
- Blockchain
    - Regular classes: define what each data structure we need is
    - Python function: define what the behavior is

BlockChain

History and current state of the block chain.

```
@dataclass
class BlockChain:
    blocks
    state
    chain_id
```

blocks

```
    blocks: List[Block]
```

state

```
    state: State
```

chain_id

```
    chain_id: U64
```

```
def state_transition(chain: BlockChain, block: Block) -> None:
    parent_header = chain.blocks[-1].header
    validate_header(block.header, parent_header)
    validate_ommers(block.ommers, block.header, chain)
    (
        gas_used,
        transactions_root,
        receipt_root,
        block_logs_bloom,
        state,
    ) = apply_body(
        chain.state,
        get_last_256_block_hashes(chain),
        block.header.coinbase,
        block.header.number,
        block.header.gas_limit,
        block.header.timestamp,
        block.header.difficulty,
        block.transactions,
        block.ommers,
        chain.chain_id,
    )
    ensure(gas_used == block.header.gas_used, InvalidBlock)
    ensure(transactions_root == block.header.transactions_root, InvalidBlock)
    ensure(state_root(state) == block.header.state_root, InvalidBlock)
    ensure(receipt_root == block.header.receipt_root, InvalidBlock)
    ensure(block_logs_bloom == block.header.bloom, InvalidBlock)

    chain.blocks.append(block)
    if len(chain.blocks) > 255:
        chain.blocks = chain.blocks[-255:]
```

- Forkchoice

```
def validate_proof_of_work(header: Header) -> None:                hide source
    header_hash = generate_header_hash_for_pow(header)
    cache = generate_cache(header.number)
    mix_digest, result = hashimoto_light(
        header_hash, header.nonce, cache, dataset_size(header.number)
    )

    ensure(mix_digest == header.mix_digest, InvalidBlock)
    ensure(
        Uint.from_be_bytes(result) <= (U256_CEIL_VALUE // header.difficulty),
        InvalidBlock,
    )
```

  - EELS doesn't handle reorgs as it assumes only receiving the canonical chain
- State implementation
  - EELS has full state implementation, incl. functions like close_state, begin_transaction, commit_transaction, state_root, storage_root etc.
- Transaction: examples incl.
  - LegacyTransaction: Atomic operation performed on the blockchain
  - AccessListTransaction: The transaction type added in EIP-2930 to support access lists
- Block

Main problems

- Require python knowledge

- Because it's an implementation, it requires spec algorithm choices
- Verbose
- Less accessible to academics

Key benefits

- Much easier to understand
- Audience
    - EELS is accessible to most programmers
    - Core EIPs often use python-style pseudocode. Now it can be actual python
- Maintainance
    - Any programmer can read Python. It's easier for anybody to contribute.
    - Implemented all the way up to Cancun
- Testable
    - Can actually sync the chain (altho very slowly)
    - Pass the ethereum/tests suite
    - Can fill tests for use in production clients
- Cool features
    - Diffs:
        - Tool that generates difference between forks
        - As each hard fork is a complete imiplementation of the Ethereum protocol. It's hard to tell the difference between forks.
    - Fuzzing
        - Tool that generates inputs randomly and feeded into clients
        - Outputs compared to all other Ethereum clients

Resources

- Adding an EVM instruction to EELS: https://www.youtube.com/watch?v=QIcw_DGSy3s
- The future of EL specs by Peter Davies: https://archive.devcon.org/archive/watch/6/eels-the-future-of-execution-layer-specifications/?tab=YouTube

Demo: Add an opcode to EELS

- Video starts at 57:00

Q&A

- How long does it take for EELS to sync the chain?
  - It took roughly 2 weeks to sync the chain. Although it's pretty slow and hard to catch up, it has the capability.
- Regarding the repo, where to find specs post Shanghai hardfork?
  - For Cancun, the repo link: https://github.com/ethereum/execution-specs/tree/forks/cancun
- Is the Engine API spec test included under EL specs?
  - API repo: https://github.com/ethereum/execution-apis
  - Fork choice: Will add English text, describing how it works.
- Difference between EL and CL specs
  - CL specs approach: start in mark down file, then render into Python
  - EL specs approach: start in Python, and render into HTML
- The biggest challenge to develop EL specs
  - Associated tooling is the hard part, incl. capability to render everything
  - Have to build a whole new rendering system and documentation tool
- Before the EIP/ ERC repo split there was a talk of proposing EIP with EELS. Do we see that in the near future?
  - You can link EELS from EIPs now. But the original vision of moving core EIPs into this repo is kinda dead.