Tutorial on Checking ERC20 with Property-Based Testing and TLA+

Difficulty: Blue trail – Medium

Author: Igor Konnov

Last update: May 28, 2022

In this tutorial, we discuss the API of the ERC20 tokens, which are commonly used in the Ethereum blockchain. This API is particularly interesting, as it has a well-known EIP20 attack vector, discussed in EIP20.

We demonstrate how one can model this API in Python and test it via stateful testing, which is popularized by property-based testing tools such as Hypothesis.

Further, we show how to specify this API in TLA+ and analyze it it with two model checkers: Apalache and TLC. Our hope is that our tutorial would bring some clarity in the relative strengths and weaknesses of these approaches.

1. Prerequisites

In this tutorial, we do not explain the basics of TLA+. If you need such a tutorial, check Entry-level Tutorial on the Model Checker.

We assume that you have Apalache installed. If not, check the manual page on Apalache installation. The minimal required version is 0.25.5.

Additionally, in this tutorial we assume that you understand property-based testing. In particular, we are using the Hypothesis framework for Python.

2. Running example: ERC20

2.1. Three methods of ERC20

As a running example, we consider a smart contract that implements an ERC20 token. To understand this example, you do not have to know much about blockchains and smart contracts. In a nutshell, ERC20 implements a protocol, in which every user holds some amount of tokens. For simplicity, we can assume that we have only three users: Alice, Bob, and Eve. For example, at some point the balances of their tokens may be as follows:

```
balanceOf["Alice"] == 3
balanceOf["Bob"] == 5
balanceOf["Eve"] == 10
```

If our users are only holding their tokens, it is a little bit boring. In ERC20, they can transfer tokens via a "transfer" transaction:

```
transfer(sender, toAddr, value)
```

By invoking a "transfer" transaction, the user `sender` transfers `value` tokens to the user whose address is stored in `toAddr`, provided that the sender has at least `value` tokens on their balance. Technically, contracts store the balances for addresses, not users, but we will be talking about users, to keep things simple.

Consider the following two transactions:

```
transfer("Alice", "Bob", 2)   # transaction A
transfer("Bob", "Eve", 6)     # transaction B
```

In the above example, Alice sends two tokens to Bob, and Bob sends six tokens to Eve. Interestingly, if transaction B is processed before transaction A, then transaction B will fail, since Bob has only 5 tokens in his account.

Things get interesting, when we consider the possibility that some of the users are actually programs (called smart contracts). Say, Eve is a smart contract. It often happens that the human users want that smart contracts do token transfer on their behalf. However, it would be a bit dangerous, if a contract could transfer an arbitrary number of tokens from the user's account. To this end, ERC20 has "approve" transactions:

```
approve(sender, spender, value)
```

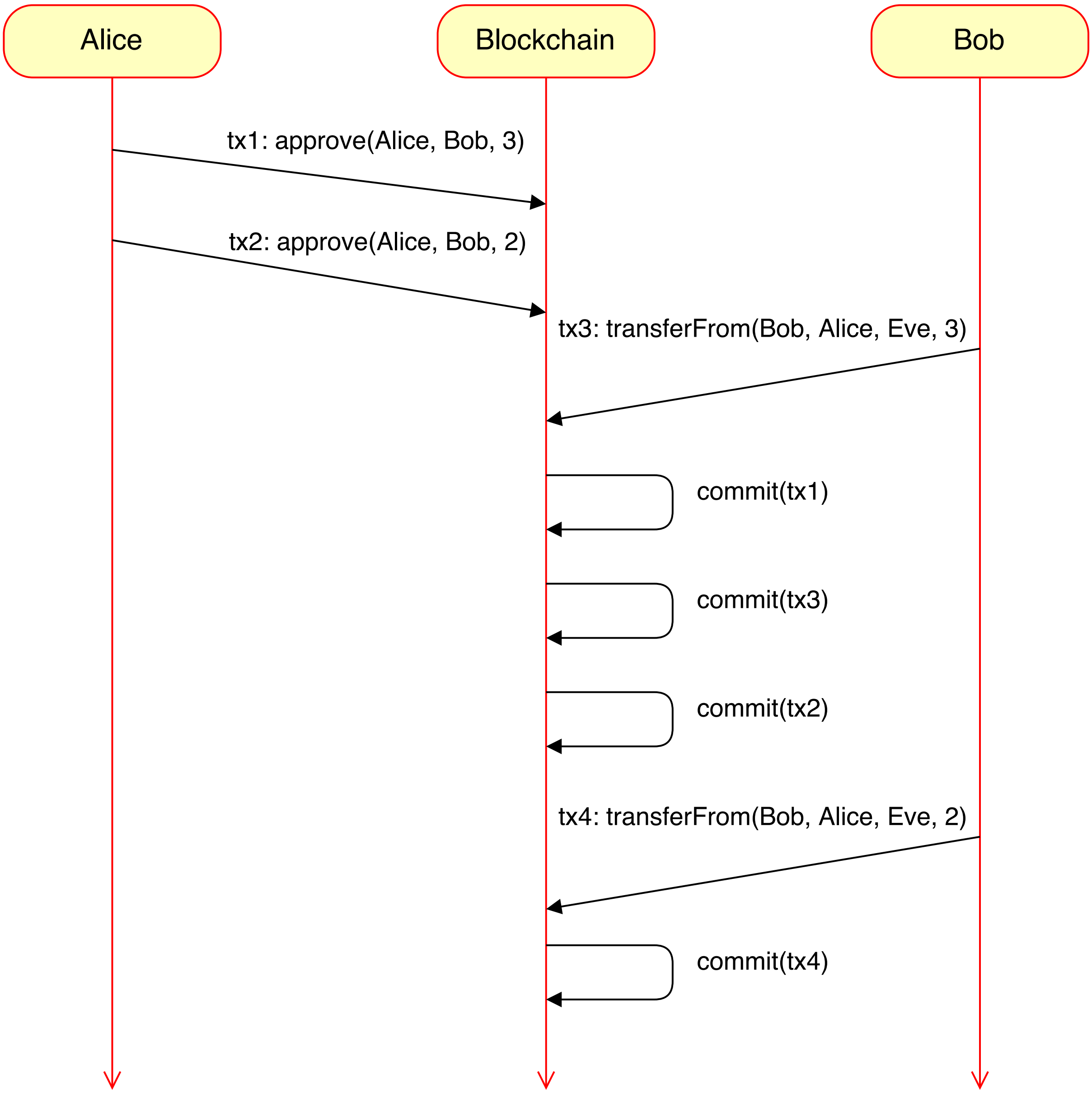
By invoking an "approve" transaction, the user `sender` authorizes the user `spender` to transfer at most `value` tokens on the behalf of `sender`. However, the spender cannot do such a transfer via a "transfer" transaction. Hence, ERC20 introduces a third type of transactions:

```
transferFrom(sender, fromAddr, toAddr, value)
```

By invoking a "transferFrom" transaction, the sender is transferring `value` tokens from the address `fromAddr` on the address `toAddr`. This can only be done, if `sender` was authorized to transfer at least `value` tokens from the address `fromAddr`.

2.2. A known issue

Although this API looks reasonable, the EIP20 attack vector shows that it may behave in a way that some users do not expect. We refer the reader to the above document for the context. Here we give a sequence of problematic transactions:


Alice
Blockchain
Bob
tx1: approve(Alice, Bob, 3)
tx2: approve(Alice, Bob, 2)
tx3: transferFrom(Bob, Alice, Eve, 3)
commit(tx1)
commit(tx3)
commit(tx2)
tx4: transferFrom(Bob, Alice, Eve, 2)
commit(tx4)

Here is what is happening in the above example. Alice approves Bob to transfer up to 3 tokens. This transaction is added to the transaction pool, but it is not committed immediately, as it takes the consensus engine some time to select this transaction and commit it. Meanwhile, Alice decides to lower her approval to Bob, and she issues another "approve" transaction that limits the amount of tokens to 2. However, Bob is actively monitoring the transaction pool, and he observes that there are two approvals issued by Alice. So he quickly issues a "transferFrom" transaction. If he gets lucky (e.g., he gives more gas to his transaction than Alice did), then his transfer happens after the first approval but before the second approval. If that happened, he issues another "transferFrom" transaction and collects five tokens in total, though Alice's intention was to authorize Bob to transfer up to three tokens (and later, even two tokens instead of three).

2.3. How to discover it?

Can we use some automation to discover such an execution? By looking at the above example, we can see that the core of this question is whether we can find the following sequence of events:

1. submit `tx1: approve(u1, u2, n)` where `n > 0` and `u1 != u2`
2. submit `tx2: approve(u1, u2, m)` where `m > 0`
3. submit `tx3: transferFrom(u2, u1, u3, k)` where `m < k <= n`, `u3 != u2` and `u3 != u1`
4. commit `tx1`
5. commit `tx3`

Once we have reached a state via the sequence of events 1-5, we can see that it should be possible to extend it with the following events:

6. commit `tx2`
7. submit `tx4: transferFrom(u2, u1, u3, l)` where `0 < l <= m`, `u3 != u2` and `u3 != u1`
8. commit `tx4`

Hence, in the rest of this tutorial, we focus on finding a valid sequence of events 1-5.

3. Stateful testing with Hypothesis

Since we are talking about an API, it is quite tempting to express this API in a programming language, for example, in Python. We give only the interesting parts of the code. A complete example is available in test_erc20.py.

3.1. Restricting the scope

Before writing the code of the transactions, we should think about the scope of our tests:

- *Do we have to run the tests against the actual blockchain?* It does not seem that we need that to reason about the API.

- *Do we have to use the actual Ethereum data structures?* Again, this is not needed for reasoning about the API.

- *Do we have to express amounts as 256-bit integers (as in Ethereum) and search over the full range of Ethereum addresses (20 bytes)?*

The last question is of particular interest, as the search spaces in modern programming languages are simply astronomic. We assume the small scope hypothesis, which is usually put like follows, e.g., at Alloy Wikipedia page:

```
...a high proportion of bugs can be found by testing a program for all test
inputs within some small scope.
```

By following this hypothesis, we limit the space of addresses and amounts to small sets (in Python):

```
# The list of addresses to use. We could use real addresses here,
# but simple readable names are much nicer.
ADDR = [ "Alice", "Bob", "Eve" ]

# We restrict the amounts to a small range, to avoid too much randomness
AMOUNTS = range(0, 20)
```

3.2. Introducing transactions

Following ERC20, we introduce three classes of transactions in Python:

```
class TransferTx:
    """An instance of transfer"""

    def __init__(self, sender, toAddr, value):
        self.tag = "transfer"
        self.sender = sender
        self.toAddr = toAddr
        self.value = value

class TransferFromTx:
    """An instance of transferFrom"""

    def __init__(self, sender, fromAddr, toAddr, value):
        self.tag = "transferFrom"
        self.sender = sender
        self.fromAddr = fromAddr
        self.toAddr = toAddr
        self.value = value

class ApproveTx:
    """An instance of approve"""

    def __init__(self, sender, spender, value):
        self.tag = "approve"
        self.sender = sender
        self.spender = spender
        self.value = value
```

3.3. Introducing and initializing the state machine

We model the API of ERC20 as a rule-based state machine. As explained in the documentation of the Hypothesis library, we introduce a class that models executions of this state machine:

```
class Erc20Simulator(RuleBasedStateMachine):
    """
    Model the behavior of the ERC20 API in terms of stateful testing.
    """

    def __init__(self):
        super().__init__()


    # more code follows...
```

The testing framework uses this state machine to randomly generate executions that are described by a set of rules, which we present below. Before, we dive into the rules, we have to initialize the state machine for every run:

```
    @initialize(amounts=gen.lists(gen.sampled_from(AMOUNTS),
                min_size=len(ADDR),
                max_size=len(ADDR)))
    def init(self, amounts):
        # balance of every account
        self.balanceOf = {
            addr: amount for (addr, amount) in zip(ADDR, amounts)
        }
        # approvals from senders to spenders
        self.allowance = {
            (sender, spender): 0 for sender in ADDR for spender in ADDR
        }
        # history variables that we need to express the invariants
        self.pendingTxsShadow = set()
        self.lastTx = None
```

The code of the method `init` is self-explanatory. The most interesting part belongs to the annotation inside `@initialize(...)`. Basically, it tells the testing framework that the input parameter `amounts` should be a randomly generated list, whose elements are randomly drawn from the list `AMOUNTS`. We limit the size of the list `amounts` with the parameters `min_size` and `max_size`. To better understand generators, check the page on the Hypothesis generators.

We have to explain Hypothesis, where to generate transactions and where to read them from. This is done via bundles. To this end, we introduce a bundle:

```
    # This bundle contains the generated transactions
    # that are to be processed
    pendingTxs = Bundle("pendingTxs")
```

3.4. Generating transactions

To generate "transfer" transactions, we introduce the rule `submit_transfer`:

```
    @rule(target=pendingTxs, _sender=gen.sampled_from(ADDR),
          _toAddr=gen.sampled_from(ADDR), _value=gen.sampled_from(AMOUNTS))
    def submit_transfer(self, _sender, _toAddr, _value):
        # submit a transfer transaction on the client side
        tx = TransferTx(_sender, _toAddr, _value)
        self.pendingTxsShadow.add(tx)
        self.lastTx = None
        return tx
```

Similar to `init`, the method parameters `_sender`, `_toAddr`, and `_value` are randomly drawn from the lists `ADDR` and `AMOUNTS`. A generated transaction of type `TransferTx` is added to the bundle `pendingTxs`, which is specified via the parameter `target` in the annotation `@rule`.

We will see later that bundles cannot be used for specifying invariants. Hence, we add the transaction to a shadow copy, which we call `self.pendingTxsShadow`. Additionally, we reset `self.lastTx`. This will be also needed for writing an invariant.

We define the rules `submit_transferFrom` and `submit_approve` similar to `submit_transfer`:

```
    @rule(target=pendingTxs, _sender=gen.sampled_from(ADDR),
          _fromAddr=gen.sampled_from(ADDR),
          _toAddr=gen.sampled_from(ADDR), _value=gen.sampled_from(AMOUNTS))
    def submit_transfer_from(self, _sender, _fromAddr, _toAddr, _value):
        # submit a transferFrom transaction on the client side
        tx = TransferFromTx(_sender, _fromAddr, _toAddr, _value)
        self.pendingTxsShadow.add(tx)
        self.lastTx = None
        return tx

    @rule(target=pendingTxs, _sender=gen.sampled_from(ADDR),
          _spender=gen.sampled_from(ADDR), _value=gen.sampled_from(AMOUNTS))
    def submit_approve(self, _sender, _spender, _value):
        # submit an approve transaction on the client side
        tx = ApproveTx(_sender, _spender, _value)
        self.pendingTxsShadow.add(tx)
        self.lastTx = None
        return tx
```

3.5. Committing transactions

To commit a generated transaction, we introduce the rule `commit_transfer`:

```
    @rule(tx=consumes(pendingTxs))
    def commit_transfer(self, tx):
        # process a transfer transaction some here in the blockchain
        # process a transfer transaction somewhere in the blockchain
        assume(tx.tag == "transfer"
               and tx.value <= self.balanceOf[tx.sender]
               and tx.value > 0
               and tx.sender != tx.toAddr)
        self.pendingTxsShadow.remove(tx)
        self.balanceOf[tx.sender] -= tx.value
        self.balanceOf[tx.toAddr] += tx.value
        self.lastTx = tx
        event("transfer")
```

The large part of the above code should be clear. However, there are two new constructs in `commit_transfer`. First, we consume a transaction via `tx=consumes(pendingTxs)`, which deletes a transaction from the bundle `pendingTxs` and places it in the input parameter `tx`. On top of that, we add the statement `assume(...)` inside the method. This statement tells the testing framework to reject the cases that violate the assumption.

Similar to `commit_transfer`, we define the rules `commit_transfer_from` and `commit_approve`:

```
    @rule(tx=consumes(pendingTxs))
    def commit_transfer_from(self, tx):
        # process a transferFrom transaction somewhere in the blockchain
        assume(tx.tag == "transferFrom"
               and tx.value > 0
               and tx.value <= self.balanceOf[tx.fromAddr]
               and tx.value <= self.allowance[(tx.fromAddr, tx.sender)]
               and tx.fromAddr != tx.toAddr)
        self.pendingTxsShadow.remove(tx)
        self.balanceOf[tx.fromAddr] -= tx.value
        self.balanceOf[tx.toAddr] += tx.value
        self.allowance[(tx.fromAddr, tx.sender)] -= tx.value
        self.lastTx = tx
        event("transferFrom")

    @rule(tx=consumes(pendingTxs))
    def commit_approve(self, tx):
        # process an approve transaction somewhere in the blockchain
        assume(tx.tag == "approve" and tx.value > 0 and tx.sender != tx.spender)
        self.pendingTxsShadow.remove(tx)
        self.allowance[(tx.sender, tx.spender)] = tx.value
        self.lastTx = tx
        event("approve")
```

3.6. Introducing state invariants

Since we are writing a test to check some properties, we have to specify these properties. The simplest property that we want to test is whether the account balances may go negative:

```
    @invariant()
    def no_negative_balances(self):
        # a simple invariant to make sure that the balances do not go negative
        # a simple invariant to make sure that the balances do not go negative
        for addr in ADDR:
            assert(self.balanceOf[addr] >= 0)
```

There is not much to explain about the above code. It is important to understand that this invariant is checked after execution of `init` and after execution of every rule in a test run.

We also write an invariant that we actually want to test:

```
    @invariant()
    def all_transfers_approved(self):
        # If this invariant is violated, then it is possible to transfer tokens
        # (based on an earlier approval), while there is an approval for a
        # smaller amount in the pending transactions
        last = self.lastTx
        if last:
            if last.tag == "transferFrom" and last.value > 0:
                for p in self.pendingTxsShadow:
                    if p.tag == "approve" \
                            and p.sender == last.fromAddr \
                            and p.spender == last.sender \
                            and last.sender != last.toAddr \
                            and p.value < last.value and p.value > 0:
                        assert(False)
```

The above invariant specifies a state that is produced by the sequence of events 1-5, as discussed in Section 2.3.

3.7. Generating the test runs

Finally, we add the test class to the test suite:

```
    unittest.main()
```

The most important parameters are as follows:

- `max_examples` limits the number of test executions to generate,
- `stateful_step_count` limits the length of test executions, and
- `deadline` limits the running time of every execution, which we set to `None`, as the running times may vary.

We run the test with the Python testing framework as follows:

```
pytest --hypothesis-show-statistics
```

We have run the test five times. Each run took 1.5 hours on average. Here is the typical output by `pytest`:

```
  - Typical runtimes: 0-3 ms, ~6% in data generation
   - 100000 passing examples, 0 failing examples, 365850 examples
   - Events:
    * 8.82%, approve
    * 1.73%, transfer
    * 0.00% transferFrom
```

Finally, on the sixth run, the test has detected an invariant violation after 34 minutes:

```
Falsifying example:
state = Erc20Simulator()
state.init(amounts=[0, 0, 2])
state.all_transfers_approved()
state.non_negative_balances()
v1 = state.submit_approve(_sender='Eve', _spender='Alice', _value=1)
state.all_transfers_approved()
state.non_negative_balances()
v2 = state.submit_approve(_sender='Eve', _spender='Alice', _value=2)
state.all_transfers_approved()
state.non_negative_balances()
state.commit_approve(tx=v2)
state.all_transfers_approved()
state.non_negative_balances()
v3 = state.submit_transfer_from(_fromAddr='Eve',
                                _sender='Alice', _toAddr='Bob', _value=2)
state.all_transfers_approved()
state.non_negative_balances()
state.commit_transfer_from(tx=v3)
state.all_transfers_approved()
state.teardown()
```

Cool! We have managed to find the expected invariant violation, though it took us about 8 hours and about 2 million runs to enumerate.

3.8. Why does it take so long?

Let's do a bit of math to better understand the probability of finding a bug with our approach. If you can propose a more precise analysis, please let us know.

As we discussed above, we have to produce a sequence that contains 5 events. Let's focus on the probability of randomly producing a run that contains five rules:

- At every step, we randomly choose one out of six rules.
    - If we choose `submit_transfer`, we have `len(ADDR) * len(ADDR) * len(AMOUNTS) = 3 * 3 * 20 = 180` combinations to choose from.
    - If we choose `submit_transferFrom`, we have `len(ADDR) * len(ADDR) * len(ADDR) * len(AMOUNTS) = 3 * 3 * 3 * 20 = 540` combinations to choose

      from.
    - If we choose `submit_approve`, we have `len(ADDR) * len(ADDR) * len(AMOUNTS)`

  `= 3 * 3 * 20 = 180` combinations to choose from.
  - If we choose `commit_transfer`, we restrict the combinations of amounts and addresses with `assume`, so this gives us a multiplier `(1 / 2) * (190 / (20 * 20)) = 0.2375`
  - If we choose `commit_approve`, we restrict the combinations of amounts and addresses with `assume`, so this gives us a multiplier `(1 / 2) * (19 / 20) = 0.475`
  - If we choose `commit_transfer_from`, we restrict the combinations of amounts and addresses with `assume`, so this gives us a multiplier `(1 / 2) * (190 / (20 * 20))^2 = 0.1128125`.

To count the number of valid runs that contain 5 rules, we execute a custom script count_combinations.py. This script gives us 591e12 combinations.

Although the search space is quite large, we have to understand the number of runs that violate the invariant `all_transfers_approved`. Maybe this number is of comparable size?

Recall that we are looking for the following sequence, which is pretty much fixed:

1. submit `tx1: approve(u1, u2, n)` where `n > 0` and `u1 != u2`
2. submit `tx2: approve(u1, u2, m)` where `m > 0`
3. submit `tx3: transferFrom(u2, u1, u3, k)` where `m < k <= n`, `u3 != u2` and `u3 != u1`
4. commit `tx1`
5. commit `tx3`

How many combinations do we have here? We see that all three addresses `u1`, `u2`, and `u3` must be distinct. Hence, the number of combinations for producing these addresses is `3! = 6`. The choice of `n` is unrestricted, so we have `len(AMOUNTS) = 20` combinations. As for `m` and `k`, we have the constraint `m < k <= n`. We can easily compute the number of the combinations for `n`, `m`, and `k` with a Python loop:

```
sum = 0
for n in range(1, 20);
    for k in range(1, n + 1):
        for m in range(1, k):
            sum += 1
print(sum)
# 1140
```

Hence, we have `6 * 1140 = 6840` runs that violate the invariant. This gives is 6840 chances in 591e12. This is about 1 chance in 86 billion, assuming the uniform distribution. We were quite lucky that Hypothesis has reported an invariant violation after exploring about 2 million runs (after exploring runs for 8 hours). Perhaps, Hypothesis is using clever heuristics to enumerate runs.

3.9. Lessons learned

Some lessons learned:

- It took us several iterations to debug the Python code, since these errors are only reported at runtime. To strengthen the model, we would have to write unit tests for the simulator.
- Since the tests take a lot of time, there is always a doubt about whether the invariants are written correctly. It is not easy to guide the framework into an interesting state.
- Random exploration is producing plenty of invalid executions (about 80% in our case), which are rejected by the framework.
- We had to carefully tune the maximum number of steps in a single run. Further increasing the number of steps would lead to further decrease in the probability of finding an invariant violation.
- Given our complexity estimates and the running times, it looks like our example is on the edge of what is feasible with Hypothesis.

4. Symbolic simulation with Apalache

Let us repeat the same exercise with TLA+ and Apalache. Although TLA+ is not a programming language, we will see that the TLA+ specification is structurally quite similar to the test that we have developed for Hypothesis. In contrast to 8 hours of running PBT, we find the same execution with Apalache in 12 seconds. So it is probably worth looking at.

We assume that you already know the basics of TLA+. The complete specification and its model checking instance can be found in ERC20.tla and MC_ERC20.tla.

4.1. The shape of the state machine

Similar to our Python code in test_erc20.py, we declare the set of all addresses. In contrast to the code, we declare `ADDR` and `AMOUNTS` as constants, which are instantiated later:

```
CONSTANTS
    \* Set of all addresses.
    \*
    \* @type: Set(ADDR);
    ADDR,
    \* Set of the amounts to use.
    \*
    \* @type: Set(Int);
    AMOUNTS
```

Since we specify a state machine, we declare the state variables of our state machine that we obviously need for ERC20:

```
VARIABLES
    \* Token balance for every account. This is exactly as `balanceOf` in ERC20.
    \*
    \* @type: ADDR -> Int;
    balanceOf,
    \* Allowance to transfer tokens
    \* from owner (1st element) by spender (2nd element).
    \* This is exactly as `allowance` of ERC20.
    \*
    \* @type: <<ADDR, ADDR>> -> Int;
    allowance
```

Similar to the Python code, we declare additional state variables:

```
\* Variables that model Ethereum transactions, not the ERC20 state machine.
VARIABLES
    \* Pending transactions to be executed against the contract.
    \* Note that we have a set of transactions instead of a sequence,
    \* as the order of transactions on Ethereum is not predefined.
    \* To make it possible to submit two 'equal' transactions,
    \* we introduce a unique transaction id.
    \*
    \* @type: Set(TX);
    pendingTransactions,
    \* The last executed transaction.
    \*
    \* @type: TX;
    lastTx,
    \* A serial number to assign unique ids to transactions
    \* @type: Int;
    nextTxId
```

4.2. Initializing the state machine

As usual, we describe the initial states via the predicate `Init`:

```
\* Initialize an ERC20 token.
Init ==
    \* every address has a non-negative number of tokens
    /\ balanceOf \in [ADDR -> AMOUNTS]
    /\ \A a \in ADDR: balanceOf[a] >= 0
    \* no account is allowed to withdraw from another account
    /\ allowance = [ pair \in ADDR \X ADDR |-> 0 ]
    \* no pending transactions
    /\ pendingTransactions = {}
    /\ nextTxId = 0
    /\ lastTx = [ id |-> 0, tag |-> "None", fail |-> FALSE ]
```

4.3. Submitting transactions

To submit a "transfer" transaction, we introduce the action `SubmitTransfer`:

```
SubmitTransfer(_sender, _toAddr, _value) ==
    LET newTx == [ id |-> nextTxId, tag |-> "transfer", fail |-> FALSE,
                   sender |-> _sender, toAddr |-> _toAddr, value |-> _value ]
    IN
    /\ pendingTransactions' = pendingTransactions \union { newTx }
    /\ lastTx' = [ id |-> 0, tag |-> "None", fail |-> FALSE ]
    /\ nextTxId' = nextTxId + 1
    /\ UNCHANGED <<balanceOf, allowance>>
```

The above code is simple. We construct a transaction as a record and add it to the set of the pending transactions.

Similar to that, we define the actions `SubmitApprove` and `SubmitTransferFrom`:

```
SubmitApprove(_sender, _spender, _value) ==
    LET newTx == [
            id |-> nextTxId,
            tag |-> "approve", fail |-> FALSE,
            sender |-> _sender, spender |-> _spender, value |-> _value
        ]
    IN
    /\ pendingTransactions' = pendingTransactions \union { newTx }
    /\ lastTx' = [ id |-> 0, tag |-> "None", fail |-> FALSE ]
    /\ nextTxId' = nextTxId + 1
    /\ UNCHANGED <<balanceOf, allowance>>
```

```
SubmitTransferFrom(_sender, _fromAddr, _toAddr, _value) ==
    LET newTx == [
            id |-> nextTxId,
            tag |-> "transferFrom", fail |-> FALSE, sender |-> _sender,
            fromAddr |-> _fromAddr, toAddr |-> _toAddr, value |-> _value
        ]
    IN
    /\ pendingTransactions' = pendingTransactions \union { newTx }
    /\ lastTx' = [ id |-> 0, tag |-> "None", fail |-> FALSE ]
    /\ nextTxId' = nextTxId + 1
    /\ UNCHANGED <<balanceOf, allowance>>
```

4.4. Committing transactions

To commit a transfer transaction, we introduce the action `CommitTransfer`:

```
CommitTransfer(_tx) ==
    /\ _tx.tag = "transfer"
    /\ pendingTransactions' = pendingTransactions \ { _tx }
    /\ pendingTransactions' = pendingTransactions \ { _tx }
    /\  LET fail ==
          \/ _tx.value < 0
          \/ _tx.value > balanceOf[_tx.sender]
          \/ _tx.sender = _tx.toAddr
        IN
        /\ lastTx' = [ _tx EXCEPT !.fail = fail ]
        /\ IF fail
          THEN UNCHANGED <<balanceOf, allowance, nextTxId>>
          ELSE \* transaction succeeds
               \* update the balances of the 'sender' and 'toAddr' addresses
            /\ balanceOf' = [
                 balanceOf EXCEPT ![_tx.sender] = @ - _tx.value,
                                  ![_tx.toAddr] = @ + _tx.value
               ]
            /\ UNCHANGED <<allowance, nextTxId>>
```

The interesting aspect here is that we mark a transaction as failed, if it violates the validation rules. Although it is not of importance in this tutorial, it is a good pattern, which lets us to produce transactions that can be used to test the actual implementation with an end-to-end testing framework such as Atomkraft.

Similar to `CommitTransfer`, we define the action `CommitApprove` and `CommitTransferFrom`:

```
CommitTransferFrom(_tx) ==
    /\ _tx.tag = "transferFrom"
    /\ pendingTransactions' = pendingTransactions \ { _tx }
    /\  LET fail ==
          \/ _tx.value < 0
          \/ _tx.value > balanceOf[_tx.fromAddr]
          \/ _tx.value > allowance[_tx.fromAddr, _tx.sender]
          \/ _tx.fromAddr = _tx.toAddr
        IN
        /\ lastTx' = [ _tx EXCEPT !.fail = fail ]
        /\  IF fail
            THEN UNCHANGED <<balanceOf, allowance, nextTxId>>
            ELSE \* transaction succeeds
            \* update the balances of the 'fromAddr' and 'toAddr' addresses
            /\ balanceOf' = [ balanceOf EXCEPT
                  ![_tx.fromAddr] = @ - _tx.value,
                  ![_tx.toAddr] = @ + _tx.value
               ]
            \* decrease the allowance for the sender
            /\ allowance' = [ allowance EXCEPT
                  ![_tx.fromAddr, _tx.sender] = @ - _tx.value
               ]
            /\ UNCHANGED nextTxId
```

```
CommitApprove(_tx) ==
    /\ _tx.tag = "approve"
    /\ pendingTransactions' = pendingTransactions \ { _tx }
    /\ pendingTransactions' = pendingTransactions \ { _tx }
    /\ UNCHANGED <<balanceOf, nextTxId>>
    /\  LET fail == _tx.value < 0 \/ _tx.sender = _tx.spender IN
        /\ lastTx' = [ _tx EXCEPT !.fail = fail ]
        /\  IF fail
            THEN \* transaction fails
              UNCHANGED allowance
            ELSE \* transaction succeeds
              \* set the allowance for the pair <<sender, spender>> to value
              allowance' =
                [ allowance EXCEPT ![_tx.sender, _tx.spender] = _tx.value ]
```

4.5. Introducing the transition predicate

As usual, we introduce the predicate called `Next` that captures the choice of actions:

```
\* The transition relation, which chooses one of the actions
Next ==
    \/ \E sender, toAddr \in ADDR:
         \E value \in AMOUNTS:
           SubmitTransfer(sender, toAddr, value)
    \/ \E sender, fromAddr, toAddr \in ADDR:
         \E value \in AMOUNTS:
           SubmitTransferFrom(sender, fromAddr, toAddr, value)
    \/ \E sender, spender \in ADDR:
         \E value \in AMOUNTS:
           SubmitApprove(sender, spender, value)
    \/ \E tx \in pendingTransactions:
        \/ CommitTransfer(tx)
        \/ CommitTransferFrom(tx)
        \/ CommitApprove(tx)
```

We non-deterministically pick one of the six actions at each step. The action parameters are non-deterministically chosen via the operator "exists", e.g., `\E value \in AMOUNTS`. Note that we simply draw integer values from the set `AMOUNTS`, as there is no need to restrict this set. Although TLA+ as a language does not have randomization, some tools may interpret non-determinism as random choice.

4.6. Introducing state invariants

Similar to `all_transfers_approved` in test_erc20.py, we define the following state invariant:

```
NoTransferFromWhileApproveInFlight ==
    LET BadExample ==
        /\ lastTx.tag = "transferFrom"
        /\ lastTx.tag = "transferFrom"
        /\ lastTx.value > 0
        /\ ~lastTx.fail
        /\ \E approval \in pendingTransactions:
            /\ approval.tag = "approve"
            /\ approval.sender = lastTx.fromAddr
            /\ approval.spender = lastTx.sender
            /\ lastTx.sender /= lastTx.toAddr
            /\ approval.value < lastTx.value
            /\ approval.value > 0
    IN
    ~BadExample
```

4.7. Introducing an instance for model checking

Our TLA+ specification is parameterized in the sets `ADDR` and `AMOUNTS`. In order to run Apalache, we have to initialize this constant. The complete code can be found in MC_ERC20.tla. The most important definitions are as follows:

```
\* Use the set of three addresses.
\* We are using uninterpreted values, similar to TLC's model values.
\* See: https://apalache.informal.systems/docs/HOWTOs/uninterpretedTypes.html
ADDR == { "Alice_OF_ADDR", "Bob_OF_ADDR", "Eve_OF_ADDR" }

\* Apalache can draw constants from the set of all integers
AMOUNTS == Int
```

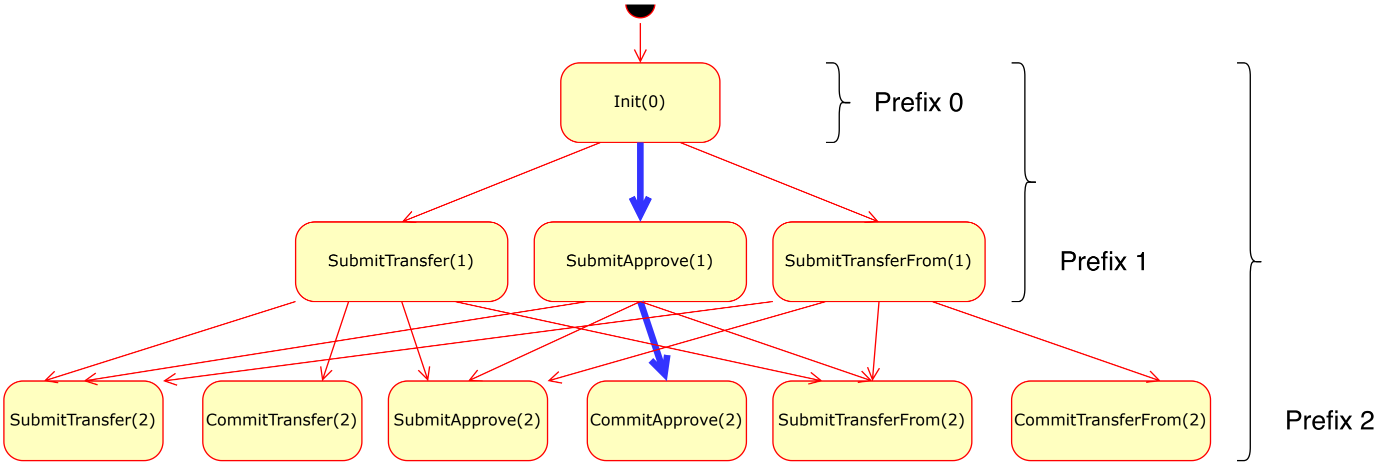
4.8. Checking the invariant via symbolic simulation

Having defined the specification and its instance, we run Apalache to perform symbolic simulation:

```
$ apalache-mc simulate --length=10 --max-run=10000 \
  --inv=NoTransferFromWhileApproveInFlight MC_ERC20.tla
...
State 10: state invariant 0 violated.
...
It took me 0 days  0 hours  0 min 12 sec
```

As we can see, Apalache came back after enumerating 33 runs in 12 seconds. You can check counterexample10.tla to make sure that it indeed violates the invariant.

It did not report the shortest execution though. This is because we have run Apalache in the simulation mode. In this mode, it randomly chooses one of the enabled actions at every step and adds it to a set of constraints that encode an execution. Whether there is an execution that satisfies the constraints is solved by the SMT solver Z3.
Consider the following figure:



Here is what is happening in the figure:

1. Apalache applies the predicate `Init`. This gives us an execution prefix of length 0, which contains only `Init`. Apalache checks the invariant for Prefix 0 as a set of constraints with Z3.

2. Apalache finds that three actions are enabled in the end of Prefix 0: `SubmitTransfer`, `SubmitApprove`, and `SubmitTransferFrom`. The model checker randomly picks the action `SubmitApprove`. This gives us an execution prefix of length 1, which is obtained by applying `Init` and then `SubmitApprove`. This gives us Prefix1. Apalache checks the invariant in the end of Prefix 1 as a set of constraints with Z3.

3. Apalache finds that there are four enabled actions in the end of Prefix 1: `SubmitTransfer`, `SubmitApprove`, `SubmitTransferFrom`, and `CommitApprove`. It randomly picks the action `CommitApprove`, forming Prefix 2. Apalache checks the invariant in the end of Prefix 2 as a set of constraints with Z3.

4. We repeat this process for Prefix 2, ..., Prefix 9, and Prefix 10. Finally, at Prefix 10, Apalache finds an execution that is described by Prefix 10 and violates the invariant `NoTransferFromWhileApproveInFlight`.

Actually, we have repeated the process described in 1-4 multiple times. In our case it was 33 times, but it may differ from run to run. We call this process *symbolic simulation*, since it combines two techniques in one:

- When we fix the sequence of actions, we perform symbolic execution. The symbolic execution is encoded in the prefixes: Prefix 0, ..., Prefix 10.

- We pick the sequences of actions at random, similar to random simulation.

4.9. How many symbolic runs do we have?

Recall that, in theory, we had to explore billions of random executions with Hypothesis, see Section 3.9. This is due to that we had to randomly pick a rule to execute as well as its inputs. With symbolic simulation, we only have to randomly pick an action, and the rest is done by the solver. Hence, we can roughly estimate the number of different symbolic runs:

1. When we limit executions to length 5, we have at most `6^5 = 7776` combinations.
2. When we limit executions to length 10, we have at most `6^10 = 60,466,176` combinations.

How many of these executions would let us discover the invariant violation? When we limit the length to 5, there is only one symbolic execution that describes exactly the sequence of events in Section 2.3. So we have 1 chance in 7776 to find the bug. If we run the simulation 7776 times, we should find it with high probability. In this example, it takes about 1 second to analyze one symbolic run. Hence, we should find this bug in about 1 hour on average. Given that we usually find it in a matter of seconds, our estimate on the number of symbolic runs is probably too pessimistic.

When we limit the length to 10, it seems that we have an unmanageable number of runs. However, recall that we have to find a run that contains the sequence of events as a subsequence! It seems that the probability of finding an invariant violation in a single run stays the same as for the length of 5. (This argument requires a more careful analysis.)

Interestingly, when we set the execution length to 50, Apalache typically finds an invariant violation in the first symbolic run after 20-40 steps in 5-7 seconds. This is probably explained by uniform randomization of actions and that multiple short runs are packed in a single long run.

4.10. Do we have to enumerate runs at all?

This is a good question. Apalache supports another mode that analyzes all symbolic runs of given length at once, without enumerating them.

5. Bounded model checking with Apalache

5.1. Finding a invariant violation

Whereas in symbolic simulation we were randomly picking a sequence of actions and delegating the discovery of right inputs to the solver, in the checking mode, we delegate the choice of the right actions to the solver too:

```
$ apalache-mc check --length=10 \
  --inv=NoTransferFromWhileApproveInFlight MC_ERC20.tla
```

```
...
State 5: state invariant 0 violated.
...
It took me 0 days  0 hours  0 min 7 sec
```

This time the model checker has found the shortest execution that violates the invariant. You can examine it in counterexample5.tla.

5.2. When there is no invariant violation

So far, the difference between `simulate` and `check` was not obvious. Their performance seems to be comparable. We can see a dramatic difference when we test an invariant that actually holds true. Consider the following invariant:

```
\* Make sure that the account balances never go negative.
NoNegativeBalances ==
    \A a \in ADDR:
        balanceOf[a] >= 0
```

Let us check all executions that make up to 10 steps with Apalache:

```
apalache-mc check --length=10 --inv=NoNegativeBalances MC_ERC20.tla
...
Checker reports no error up to computation length 10
It took me 0 days  0 hours  9 min 32 sec
```

Now we know that, no matter what, the *invariant holds true on all states that are reachable via at most 10 steps*.

As we discussed in Section 4.9, we have to check about 7776 symbolic runs, to get a high probability of exploring all executions:

```
apalache-mc simulate --length=10 --max-run=7776 \
  --inv=NoNegativeBalances MC_ERC20.tla
...
Checker reports no error up to computation length 10
It took me 0 days  0 hours 26 min 50 sec
```

Although, it took Apalache longer, it has enumerated 7.7k symbolic runs. However, the important difference between `simulate` and `check` is that `simulate` does not give us an ultimate guarantee about all executions, even though we limit the scope to all executions of length up to 10, whereas `check` does.

5.3. What about longer executions?

As we have seen, Apalache can give us a guarantee about all executions of predefined length. What if we want to analyze all possible executions? This is harder. We refer the reader to the section on Checking an inductive invariant. We will write another tutorial on this topic.

6. State enumeration with TLC

So far we have been using Hypothesis and Apalache. Since we have a TLA+ specification, we can easily run the standard explicit model checker TLC too.

6.1. Setting up TLC

Before you can run TLC in the command-line, you have to download it:

```
wget https://github.com/tlaplus/tlaplus/releases/download/v1.8.0/tla2tools.jar
```

We have to define two auxiliary files (created with TLA+ Toolbox):

- a configuration file MC_tlc_check.cfg:
    - we define `ADDR` as a constant set of three model values: `{A_Alice, A_Bob, A_Eve}`
    - we define `AMOUNTS` to be a fixed range of `0..19`, similar to our Hypothesis tests.
    - further, we inform TLC that it can use symmetry reduction over `ADDR`.
- a model file MC_tlc_check.tla

6.2. Simulation with TLC

TLC has a built-in simulation mode, which is randomly producing traces. In our case, it can be run like that (set `$APALACHE_HOME` to the directory where Apalache is installed):

```
java -DTLA-Library=$APALACHE_HOME/src/tla -jar tla2tools.jar \
  -config MC_tlc_check.cfg -simulate num=1000000000 -depth 10 MC_tlc_check.tla
...
Error: Invariant NoTransferFromWhileApproveInFlight is violated.
...
The number of states generated: 1843943938
Progress: 1843943943 states checked, 183661963 traces generated (trace length:
mean=10, var(x)=0, sd=0)
Finished in 01h 09min at (2022-05-27 17:52:48)
```

As we can see, TLC has found an invariant violation after 1 hour, though it is a matter of luck, since the simulation is done at random. Interestingly, TLC is enumerating traces faster than Hypothesis did in our experiments with test_erc20.py, see Section 3.7.

Interestingly, *when we set the search depth to 30 or even 50, TLC finds an invariant violation in*

Interestingly, *when we set the search depth to 30 or even 50, TLC finds an invariant violation in less than a minute*. On the other hand, when we set the depth to 5, it enumerates the maximum number of runs without hitting a bad state. Understanding this relation between the number of steps and the time to find a bug needs further research.

6.3. State enumeration with TLC

Similar to model checking with Apalache, we can run TLC to check the invariant via state enumeration:

```
java -DTLA-Library=$HOME/devl/apalache/src/tla -jar tla2tools.jar \
  -config MC_tlc_check.cfg -nworkers 4 -fpmem .75 MC_tlc_check.tla
```

Note that we let TLC to use 75% of the available memory and ran it on 4 CPU cores (make sure you have them or change this setting!). Our experiments server ran out of disk space (100 GB) after 1 hour and 20 minutes. TLC has produced 738 million distinct states, most of them left on the search queue. By that time, it has reached the diameter of 3, whereas it would need the diameter of 5 to find an invariant violation.

7. Conclusions

We summarize our findings in the following table.

We only ran one experiment for each row in the table. To get a better understanding of the average running times, we would have to perform each experiment multiple times. Hence, take these figures as one observation, not as a hard fact.

Input	Tool	Method	Performance bottleneck	Complete?	Tim expe
Python PBT	Hypothesis	Property-based testing, stateful testing	combinatorial explosion of executions	no	8
TLA+	Apalache	Symbolic simulation	combinatorial explosion of symbolic executions & SMT	no	5 s `len` 12 `len`

Input	Tool	Method	Performance bottleneck	Complete?	Tim expe
TLA+	Apalache	Bounded model checking	combinatorial explosion in SMT	yes: for fixed length and fixed parameters	7
TLA+	TLC	Explicit enumeration + simulation	combinatorial explosion of executions	no	< 1 `dept` hou for `d`
TLA+	TLC	Explicit model checking	combinatorial explosion of states	yes: for fixed parameters	>1.5 disk re diar

Since we have conducted the experiments on one benchmark only, we are not trying to draw general conclusions from this table. However, we propose some intuition about why the tools behaved this way:

- Stateful testing with PBT is randomly choosing rules and their inputs. Hence, in theory, this should be the worst case of combinatorial explosion in our experiments. Nevertheless, the tool has found an invariant violation. We do not know, whether it was sheer luck or clever heuristics of Hypothesis. Interestingly, increase the number of steps did not help us in finding an error faster, in contrast to TLC and Apalache.

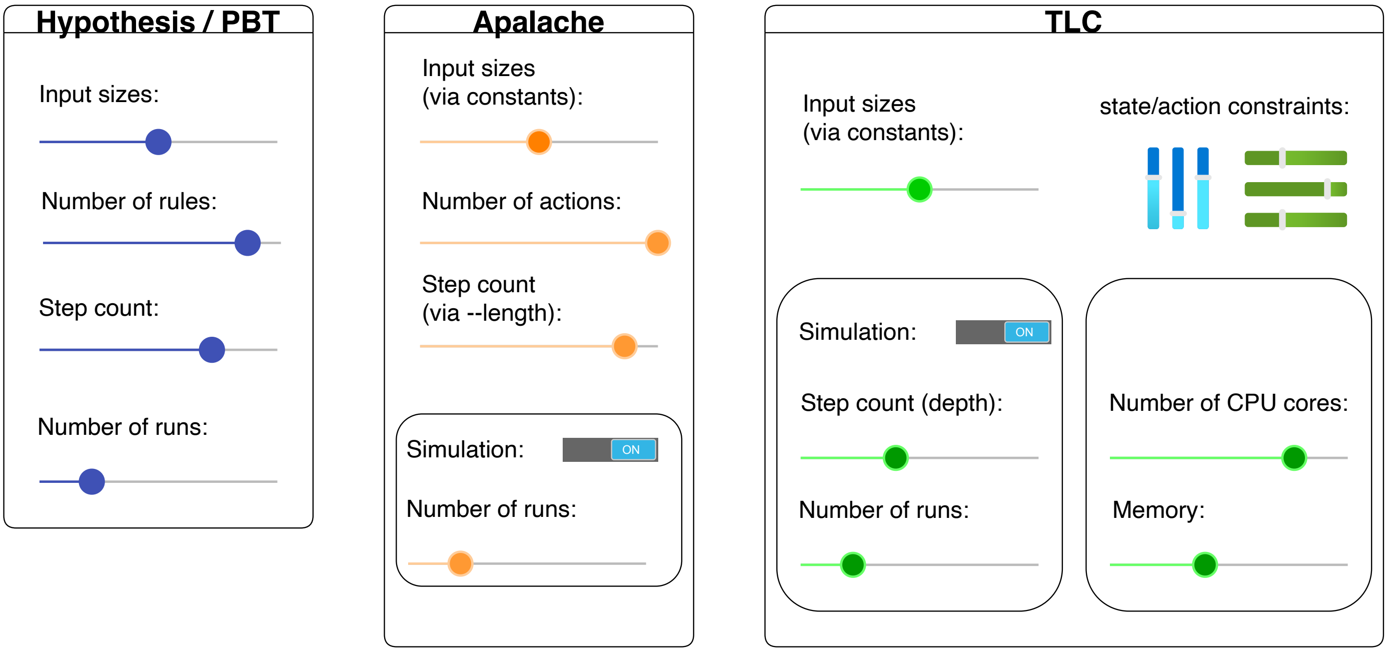
- Symbolic simulation with Apalache was very quick at finding an error. This is due to the fact the number of *symbolic runs* is growing much slower than the number of *concrete runs* in this example. Interestingly, when we increase the number of steps, Apalache finds an invariant violation even faster.

- Bounded model checking with Apalache was the fastest one. This should not come as a surprise, as we are using the SMT solver Z3. The SAT/SMT community have been tackling NP-complete problems and combinatorial explosion for decades. Hence, SMT is better tuned to the search problem than an ad-hoc random exploration. As we have seen, this mode slows down, when there is no error.

- For the depth of 50, both TLC and Apalache found an invariant violation very quickly (less than a minute and 5 seconds, respectively). For depth of 10, TLC simulation was dramatically slower than the Apalache simulation. The number of generated traces in 1 hour was significantly larger than the number of traces produced with Hypothesis. It would be interesting to see why this is happening. We conjecture that the simulation technique of TLC has a uniform distribution over successor states, not over the enabled actions.

- Explicit state enumeration with TLC was extremely slow. This is not very surprising, as TLC has to deal with relatively large state spaces here. Since TLC implements breadthfirst search, it has to enumerate a massive number of states, before it increases the depth.

It is also important to understand all kinds of controls that we have over the search process in the tools. For instance, removing the "transfer" transaction would significantly reduce the size of the search problem, and it is safe to do so for checking `NoTransferFromWhileApproveInFlight`. The following figure summarizes all kinds of controls that we have found in Hypothesis, Apalache, and TLC.



The most important controls are the size of the inputs and the number of actions/rules. These parameters are under control of the specification writer, and they affect the search problem the most. The second most important controls are those that control the scope of the search such as the number of steps and the number of runs to try. Both Apalache and TLC allow the user to switch between simulation and classical model checking. Simulation is typically much faster and scales much better with a larger number of steps in a run. However, simulation is inherently incomplete and requires some pen & paper reasoning to understand the achieved coverage, as we have done in this tutorial. Classical model checking modes come with proven guarantees of completeness, though these guarantees vary depending on the implemented technique. Finally, TLC has extensive controls on the number of cores and memory usage. Although these controls do not change the size of the problem, they may help one to get the tool feedback faster.

If you are curious, we have tried to push the parameters of Hypothesis and TLC to the absolute minimum, e.g., by setting `AMOUNTS` to `0..2` and restricting the number of steps to 5. This have not led to a significant improvement in performance.

In conclusion, we believe that all these methods and tools have their place in the developer's tool belt. However, as with all advanced tools, we have to understand, where they fit in the development process and what can affect their performance and completeness. For instance, the Apalache team is using property-based testing framework Scalacheck to find hardcore bugs in the model checker itself. We call this tool Nitpick.

Disclaimer: Although we are expert users of Apalache and TLC, we are beginners in Hypothesis. If you know how to improve our experience with Hypothesis, please let us know. A pedantic reader will notice that we have chosen Python, which is probably not the most performant programming language. We chose Python for its simplicity and relative popularity. One can probably achieve much better performance with Rust or Golang. If you would like to contribute a property-based test similar to test_erc20.py and contribute the experimental results to this tutorial, please let us know. We will be happy to include them in this tutorial.