

ARTICLE TYPE

Testing feature-rich blockchains

Thomas Arts¹ | Hans Svensson¹ | Clara Benac Earle^{*2} | Lars-Åke Fredlund²¹Quviq AB, Gothenburg, Sweden²Universidad Politécnica de Madrid, Spain**Correspondence**

*Clara Benac Earle, UPM Campus de Montegancedo s/n, Boadilla del Monte 28660, Spain, Email: cbenac@fi.upm.es

Blockchain implementations have become more and more advanced, combining many different features in the same framework (oracles, names, state channels, etc.). Since the cost of errors in reputation and represented value is high in the blockchain world, software quality is of the utmost importance. One of the main methods used to assure such high software quality is careful testing. However, the number of tests needed to achieve a high level of assurance grows quadratic with the pairs of features of the blockchain, and when testing triples of features the growth is cubic. To manually craft the required large number of tests is an almost impossible undertaking in practice. In this paper, we describe how property-based testing techniques have been used to automate testing of the core part of the Aeternity blockchain, ensuring the high software quality of the blockchain. [Even though property-based testing is a powerful testing technique, applying it to the task of testing a complex system such as a blockchain is far from trivial. The structure of the Aeternity property-based test model follows the structure of the blockchain, i.e., it cleanly separates different blockchain features \(e.g., oracles, smart contracts\) into different model parts, and moreover, reduces the amount of boilerplate test model code by focusing on the identification of valid blockchain transactions. The test model is evaluated through a careful instrumentation of test code which permits observations of which combinations of features have been tested during a test run, and with which frequency. The article documents the details for how these issues were addressed in the development of the Aeternity test model, providing insights into both the testing of other blockchains as well as the testing of other complex feature based systems.](#)

KEYWORDS:

property-based testing, blockchain

1 | INTRODUCTION

Blockchain technology has been attracting a lot of attention since the first blockchain was proposed by Nakamoto¹ in 2009. Many blockchain platforms are being developed in a race to become “the most successful blockchain”. Some examples are Bitcoin², Ethereum³, Tezos⁴, and Aeternity⁵. In this competition, blockchain platforms must show that they are reliable, secure, scalable, efficient, etc. while providing sophisticated features that set them apart from the competition like smart contracts, oracles, and a naming system. For instance, Ethereum was the first blockchain to focus on smart contracts, Bitcoin provides a naming system via Blockstack⁶.

In this paper, we report on our practical experience applying PBT to test a particular blockchain platform, Aeternity⁵. Concretely, we have tested with PBT the transaction engine of Aeternity which is implemented in the functional programming language Erlang⁷, which has its origins in the telecommunication industry, and is nowadays used in many large internet “back-end” systems, e.g., WhatsApp, bet365, etc. Although this study is limited to Aeternity, the approach is applicable to other blockchains as they share many design characteristics with Aeternity. That is, they implement feature based transaction engines. The details are of course different, e.g., with respect to blockchain implementation language, and concerning the choice of a smart contract language, but that does not affect our testing approach much. The test tool we use, Quviq QuickCheck⁸, permits testing software written in non-Erlang programming languages (see⁹ for the description of a project testing a large non-Erlang code base). Moreover, for testing a particular transaction engine we do not need to consider the exact semantics of the smart contract language, but rather how contracts are invoked, how contracts calls are initiated, and how contracts are terminated. That is, interactions that happen in similar ways in most blockchain implementations.

The transaction engine which we have tested comprises around 15,000 lines of Erlang code implementing the core logic for the interaction of blockchain users with the Aeternity blockchain. For example, the “spend” transaction transfers some amount of cryptocurrency between two accounts as an atomic action. In total, the developed PBT model considers 18 different transactions, providing basic services such as cryptocurrency transfer (“spend” transaction) as well as more advanced blockchain “features” (e.g., smart contracts, oracles, a symbolic naming system for accounts, etc.). The blocks in the Aeternity blockchain should be composed of sequences of *valid* transactions, i.e., transactions that are considered to be valid according to the agreed upon Aeternity consensus protocol. As an example, to be considered valid, a spend transaction must specify an origin account that already exists and has sufficient funds, but the destination account need not exist. Note that the consensus protocol may change over time, as e.g. new transaction types might be admitted to the blockchain. That is, a blockchain block b_1 can have been admitted by a consensus protocol p_1 , whereas a later block b_2 can have been admitted by a different consensus protocol p_2 . The consensus protocol that applies to a particular block is determined by the height of the block in the blockchain.

Concretely then, our goal is to test that the Aeternity *implementation* respects the *specification* of the Aeternity consensus protocol, i.e., that the implementation only includes a new transaction in a block (at blockchain height h), if the consensus protocol in place at height h deems the transaction valid.

Blockchains are especially interesting as items of study from the point-of-view of testing as there is an understandable emphasis on reliability and availability concerns, while at the same time blockchain implementations must permit rapid change. The emphasis on reliability is easy to understand given that design or programming mistakes can be very costly in terms of money or reputation lost. In particular, once a transaction has been accepted as valid by the blockchain, the transaction can never be deleted and must forever be considered valid. At the same time, there is quite a lot of competition between blockchain implementations for marketplace too, leading to pressure to offer more convenient platform features, better smart contract languages, and so on, with the consequence that parts of the basic blockchain platform software can undergo rapid changes and additions. To complicate the situation even more, the Aeternity software may not be updated on all nodes at the same time. Given that, it is no surprise that all blockchain platform developers devote a lot of effort to test their blockchain platform. In the case of Aeternity, for instance, there are close to 600 unit tests and over 600 integration tests of which a large part is concerned with testing that the implementation respects the consensus protocol, both by testing that valid transactions are accepted and that invalid transactions are rejected. The manual crafted tests, used by developers as well as in continuous integration, have shown to be reasonably effective in catching errors before the code gets into production. However, the number of tests grows linearly with the number of fields that can be valid or invalid in each transaction, thus, having 200 tests of this kind would cover most single feature cases.

In addition, different blockchain transactions may interact in many possible ways, desired or not. For instance, a spend transaction may specify “symbolic” account names, which should have previously been registered using another transaction type. Thus, typical feature interaction tests are part of the common test suite, but the number of tests required grow quadratic with the number of transactions defined. As an example let us consider the transaction that implement the “oracles” blockchain feature. Intuitively, an oracle enable agents external to the blockchain to provide information to blockchain users through a transaction which queries the oracle, and a transaction which provides a response from the oracle to such a query. Such oracles queries can for instance be used to import e.g. external financial data to the blockchain. Thus, given that there are tests for querying an oracle and responding to a query (most likely one and the same test case), then one would like to interleave this test with spend transactions to the oracle. A spend from the oracle has two interesting cases, enough funds left to answer and not enough funds left to answer. One can do the spend before or after the query, but it should happen before the response. As we have seen, by just considering some interesting cases when spend and oracles are combined, we have identified a number of additional tests that would be very worthwhile. If we, for instance, adopt the strategy of writing one test case for each pair of transactions, another

400 tests are needed. Clearly, many of those could be discarded because implementation knowledge would disqualify some tests from being interesting, but that is a dangerous slope to walk.

Moreover, testing only pairs of transactions is often not enough. There are interesting tests that contain at least 3 different transactions, i.e., in particular when testing a particular feature (e.g., names, oracles, etc.). For instance, the Aeternity test code includes a test case which first pre-claims a symbolic account name, then claims it, and finally revokes it. Ideally, we should interleave the transactions in this test with other tests testing other features, e.g., the oracle feature.

In summary, with a large number of transaction types that can, potentially, interact in both desired and undesired ways, the task of writing, manually, unit test cases to check such interactions is a serious challenge.

Typically, a tester will write unit tests for all possible combinations of transactions she/he can think of, with the risk of missing some potentially crucial test. A complementary approach is to [automatically generate random test cases using the testing technique known as property-based testing \(PBT\)](#) ^{10,11,8}. Such automatic generation of random test cases from the domain of all possible test cases has been demonstrated to be an effective method to find software errors ^{9,12,13}.

Using traditional testing would require to write a large number of unit tests, due to the need to test for wanted or unwanted interactions between the different features of the blockchain, i.e., so called [feature interactions](#) ¹⁴. For instance, the smart contract transactions may interact closely with spend transactions, and with the oracle transactions, and with naming transactions, as smart contracts may refer to oracles, and may be named. Thus, we need unit tests that execute these types of transactions in arbitrary orders, and decide what the correct execution result is. This quickly becomes a very tedious exercise; in the worst case the number of desirable unit test cases will grow exponentially in the number of features of the blockchain. This is the case, not only for Aeternity, but for most blockchain that include such features.

In PBT, instead of writing individual test cases, one writes higher-level test properties, from which an unlimited number of individual, random, test cases can be automatically generated. As we shall see in this paper, applying this approach to testing a blockchain transaction engine requires us to write one test description (consisting of a couple of functions) per transaction type; currently there are around 20 such transaction types in Aeternity.

During the development of the Aeternity blockchain, property-based testing has not only been used for testing transactions but also for testing many other functionalities in the system, such as encoding/decoding, serialisation, compiling smart contracts, etc. However, in this article we focus on the problem of testing transactions, as we believe the findings are largely applicable to other blockchains too, as many share similar features (e.g., oracles, name register, etc.). Moreover, testing transactions well is a challenging and important task, in that we must systematically test for both wanted and unwanted feature interactions.

Although the article focuses on Aeternity, the testing techniques used permit application to other blockchains such as Ethereum with little substantial changes, e.g., in practice we have to modify test models to take into account the variations in what features are available, account for differences in transaction types, and the manner in which transaction costs are calculated, etc. [In fact, we have already applied this approach to testing some smart contracts written in Solidity and running in the Ethereum blockchain platform](#) ¹⁵. [In this paper, we give some recommendations for testing blockchain platforms using PBT.](#)

A vital ingredient in a successful test machinery is being able to judge how good a test run is, i.e., [did we achieve enough testing?](#) Traditional methods to measure whether this is so involve calculating different coverage measures over the system-under-test, such as line coverage or path coverage ¹⁶. PBT supports these measures too, but also permits a tester to annotate the test code with the “features” being tested, which permits a coverage analysis based on system specific coverage measures. Using such an approach we can, for instance, calculate what percentage of “spend” transactions in a test run were invoked with incorrect parameters (e.g., non-existent accounts, or lack of funds, etc.).

The organisation of the rest of the article is as follows. In Sect. 2 we give a general introduction to blockchain technology, and Sect. 3 describes the particular details of Aeternity blockchain in detail. The property-based testing technique is explained in Sect. 4. In Sect. 5 we explain how we used property-based testing for the Aeternity blockchain by highlighting the new ideas. The results of testing the transaction engine of Aeternity are summarised in Sect. 6 [In Sect. 7 we discuss lessons learnt, and give some recommendations on how to apply PBT techniques to testing blockchain platforms.](#) Related work is described in Sect. 8. Finally Sect. 9 summarizes the approach, and discusses future work.

2 | BLOCKCHAIN PLATFORMS

In blockchain technology, the central idea is to have a shared ledger which records information agreed upon by several parties. This shared ledger is implemented as a chain of blocks, a blockchain. A key characteristic of blockchain is that blocks are immutable, i.e., their content can never be changed. This is ensured by using Merkle Trees¹⁷.

In a typical blockchain platform there are transactions and participating nodes, i.e., CPUs. Transactions are atomic actions, for instance, a typical transaction is Bob sends Alice the amount X of some cryptocurrency. Typically, to execute a transaction, the sender will have to pay a transaction fee.

As a blockchain is typically composed of a set of participating peer nodes, in order to create a unique “valid” blockchain, all participating nodes have to agree upon what is a valid blockchain, and in how to add new blocks to the blockchain. Such an agreement is described in a so called consensus protocol which consists of: (i) a set of the pre-agreed rules for transactions, for instance, only accepting transactions in blocks if the transaction initiator has enough funds, and (ii) a consensus mechanism.

A blockchain block then contains an ordered set of valid transactions, the hash of the preceding block, the hash of the current block and other data like a timestamp, etc. A valid transaction is a transaction which follows this set of pre-agreed rules. A valid blockchain contains only valid blocks, i.e., blocks which only contain valid transactions. As blockchain participants can access the whole chain, whether a block is valid can be checked by (in principle) any blockchain participant.

The participating nodes that collect transactions and create blocks are typically called miners. They try to create new blocks, by adding new transactions, logically placed on top of the latest block in the blockchain. As many blockchains permit competition for creating new blocks, which block is eventually added as the next block to the chain is resolved using an agreed upon consensus mechanism. Examples of consensus mechanisms are proof-of-work (PoW)¹ and proof-of-stake (PoS)¹⁸.

Time in a blockchain is measured in block height where the first block, the so-called genesis block, is at height zero. In a blockchain, the pre-agreed rules may change over time meaning that, up to a certain height, a valid transaction is a transaction that follows one set of pre-agreed rules while, after that height, they may follow a different set of pre-agreed rules. This is a key characteristic of any blockchain: blocks in the blockchain are immutable, i. e., the transactions included in the block cannot be changed. **However**, the rules used to validate transactions may vary over time, i.e., **the validity of a transaction depends on the height of the block**.

2.1 | Blockchain features

Initially, Bitcoin was developed to allow payments to be sent directly from one party to another without going through a financial institution. Later on, many blockchain platforms, included more sophisticated features like the following:

- A naming system for registering names such that one can send money to a named account instead of a cryptic address. In Bitcoin, a naming system is provided by Blockstack⁶ while, in Ethereum, the Ethereum Name Service¹⁹ (ENS) provides a similar functionality.
- Smart contracts, which are reactive programs that can receive and transfer cryptocurrency and perform more general tasks with the data on the blockchain, keeping state and putting results of the computation back on the chain. An example contract could be a prediction betting service, where a blockchain participant creates a new (named) contract that is used for betting on the amount of rain during a month in Madrid. Then, contract users add their bets (predictions), and transfers cryptocurrency to the contract as payments for the bets. When the betting time is over (a month has passed), the user with the most accurate prediction is announced as the winner. The winner then receives a percentage of the cryptocurrency collected in bets, and the contract creator takes a commission. Each blockchain platform typically defines its own smart contract language, for instance, Solidity is the contract language of Ethereum while Michelson is the contract language of Tezos and Sophia is the contract language for Aeternity.
- Oracles, which are ‘trusted’ third party services that provide data external to the blockchain. For instance, in the previous example, an oracle provides to the blockchain the externally collected data concerning the amount of rain in Madrid in the past month. As an example, Reality Keys²⁰ pipe Internet information into Ethereum.

In blockchain, features are typically implemented as blockchain atomic transactions, for example, a very common transaction permits transferring money from one account to another. Other transactions are used, for instance, to create smart contracts, to define an oracle, or to register a name.

3 | THE AETERNITY BLOCKCHAIN PLATFORM

Aeternity^{5,21} is an open source blockchain platform inspired by Ethereum with the goal of improving Ethereum scalability issues. The framework runs as a decentralised trustless distributed ledger with Proof-of-Work (PoW) consensus²². For reasons of performance Aeternity uses the Bitcoin-NG²³ blockchain protocol, which splits blocks into a sequence of so called “micro blocks” that contain the transactions, and which are computationally cheap to build. Similar to Ethereum, smart contracts can be created and called. Aeternity also has a number of popular services as native transactions on the blockchain such as oracles, names and state channels. Aeternity is open source; the full source code together with documentation e.g. of the blockchain protocol is available at <https://github.com/aeternity>. Below we describe the transactions supported by the Aeternity blockchain.

3.1 | Accounts

As it is the case with most blockchain platforms, the most basic transaction is used to transfer “tokens” (cryptocurrency) from one account (identified by public keys) to another, and which is signed by the sending party. In Aeternity, such transaction is called a **spend transaction**. The spend transaction also serves as the basis for creating new accounts, by accepting any new recipient public key as a new account.

Posting a transaction to the chain can fail for many reasons, among which that the transaction is not correctly signed. But even correctly signed transactions may fail to become part of the chain.

Each transaction specifies a *fee* that the miner will eventually collect when including the transaction in a micro-block, and a “time-to-live” (ttl) value specifying at what time the transaction expires if not yet accepted by any miner. Times in transactions are specified by referring to the height of the blockchain (the number of blocks), as the chain grows monotonically (and the rate of growth is known).

The provided mining fee can be “unattractive”, in which case the transaction will stay in the transaction pool until it expires. It may also happen that the provided mining fee is lower than the minimum required fee defined by consensus, in which case it is an erroneous transaction that will be rejected (in the unlikely case the miner would find the fee attractive enough). Including a transaction that is by consensus to be rejected, such as a transaction with lower fee than the lowest agreed upon, is called fraud. Any micro-block containing such erroneous transaction is invalid and should be rejected by all other nodes in the network¹.

Each transaction also includes a *nonce* to prevent replay attacks²⁴. Each new account starts with nonce 1 and as soon as a transaction with that nonce is accepted on chain, only transactions with exactly nonce 2 are accepted, etc. A user can post a number of different transactions with the same nonce in which case it is non-deterministic which of these transactions will result on chain, but only one of them will be accepted, and the others will thereafter be rejected. This can be used as a feature by re-posting a transaction with an unattractive fee with the same nonce and a higher fee. Similarly, a user can post a whole series of transactions with increasing, but too large nonces. Only when the missing nonce is posted, all other transactions that possibly remained in transaction pools are enabled.

3.2 | Smart contracts

Smart contracts permit well-specified and well-documented interactions between blockchain participants. A contract is in essence an account (to store funds) together with a set of programmed operations. These operations can store information visible to all parties on the blockchain, and can perform transfers of cryptocurrency to and from the associated smart contract account. Aeternity smart contracts are written in the high-level blockchain smart contract languages Solidity (used in Ethereum) or the new contract language Sophia implemented through compilation to virtual machine (VM) byte code.

As an example of a contract, a user may auction an object in the real world by creating and posting a contract to the blockchain, using a **contract create transaction**. Let us assume that this is a Dutch auction, then the initial price would be set high and for each new key-block that is mined (representing the passing of time) the price is decreased. A blockchain participant can buy the object by calling a bid function implemented by the smart contract using a **contract call transaction**. When the bid function is called, the contract code computes the price for the current blockchain height. If the caller has supplied sufficient funds in the call, the seller is paid, and the bidder is charged, and the contract enters a non sell-able state for the object. The next bidder will fail the call, and will only pay for the cost of executing the transaction, but will neither pay for nor acquire the object.

¹Technically posting invalid micro-blocks is sometimes possible and a proof-of-fraud transaction is posted later to punish the node posting this transaction.

```

contract DutchAuction =
  record state = { amount : int,
                  height : int,
                  dec    : int,
                  sold    : bool }

  public function init(price, decrease) : state =
    { amount = price,
      height = Chain.block_height,
      dec    = decrease,
      sold    = false }

  public stateful function bid() =
    if (state.sold) abort("sold")
    let price =
      state.amount - (Chain.block_height - state.height) * state.dec
    if (Contract.balance < price) abort("no money")
    Chain.spend(Contract.creator, price)
    Chain.spend(Call.origin, Contract.balance)
    put(state{sold = true})

```

FIGURE 1 Sophia smart contract

The contract languages, and hence the evaluation in the virtual machines, must have access to blockchain primitives like the height of the chain, and the caller accounts. Typically, in Aeternity all blockchain primitives (transactions) are available from within a contract. The complete code for a Dutch auction is presented in Fig. 1. *Note that the contract create transaction only contains (i) the version number of the virtual machine to use for running the smart contract bytecode, and (ii) the compiled bytecode for the smart contract. That is, the high-level source code for the contract is not present in the transaction.*

The transparency of the blockchain guarantees that it is verifiable that the first valid bid on chain² is correctly paying the right price. Moreover, it is verifiable that the bidding call transactions accepted later are only charged a transaction fee and the cost of execution. The sale conditions are transparent; whether the actual object ever arrives is outside the scope of the blockchain.

Clearly there are plenty of mistakes one can make in the code of the contract. This paper is not about testing the actual code in the contract, but about testing general assumptions for contracts created and called. One such property which is vital to ensure a “healthy” blockchain is that the costs for creating and executing contracts are calculated in a “fair” and transparent manner, according to well-defined rules, which testing should check are enforced correctly. In general these rules should permit both miners to make a profit (as an incitement to include such transactions in blocks), as well as providing a cost model which is predictable for contract callers (as an incitement to enter into contracts).

For example, a contract call comes with a number of parameters *fee*, *gas* and *gas_price*, and an *amount*. The fee is a payment made to a miner to account for the costs of the inclusion of the transaction in the chain, which depends on e.g. the size of the call transaction. Actually executing the contract code invoked in a contract call requires an additional computation (or storage) effort, which is measured as the amount of “gas” spent; each instruction in the code of a contract operation has an associated cost in gas. The gas parameter, then, is an upper limit on the amount of gas the caller of the contract is willing to expend during the call, and *gas_price* is the price for a unit of gas the caller is willing to pay. Thus the maximum amount of funds a miner may charge for a call is “*fee* + *gas* * *gas_price*”. Finally *amount* are the funds sent by the participant to the contract as part of the contract call, e.g., in the dutch auction the *amount* corresponds to the payment for the object.

The minimal valid value for the call fee parameter is equal to the fixed gas costs for the transaction (e.g., due to call overhead and storage for the transaction) multiplied by the minimal permitted gas price. The minimal gas price is agreed upon by the members of the blockchain, i.e., it is under consensus.

²The blockchain does not guarantee that the first one posting a valid bid becomes the first one on chain.

The fee is never returned to the caller, even if the transaction is aborted. In contrast a miner will charge only for the actual costs (e.g., *gas* used * *gas_price*) incurred during the execution of the contract. A call may be aborted if some programmed precondition fails, or if the execution of the contract code requires more gas than the gas limit specified. In such cases the fact that the call failed is still recorded on the chain, and the fee and the gas actually consumed is charged to the caller. However, the *amount* is returned to the caller.

Another testing challenge for contracts is demonstrating that the blockchain copes with changes in the contract runtime environment. Since over time the virtual machine (VM) is enhanced and features are added to it, the contract create transaction specifies on which VM the code should run. It is important to test that contract calls intended to run on older VMs work as before, including error behaviours, when the VM software is updated.

In addition consensus can establish that from a certain height, calls to contracts using an old VM are no longer allowed. The testing code checks that such decisions are enforced, i.e., that above a certain height, calls to old contracts are rejected.

3.3 | Oracles

In order to embed information from the outside world in the blockchain, an “oracle” abstraction is used. Oracles can be used, for instance, to add sensor data or financial data to a blockchain, and they have proven to be particularly useful in combination with smart contracts.

An oracle is created, using the **oracle register transaction**, by some blockchain participant for the purpose of answering queries from (other) participants. Other blockchain participants may query the created oracle for information using the **oracle query transaction**, supplying funds to pay for the result; this creates an oracle query object on the blockchain. The oracle responds to such queries using the **oracle response transaction**, which adds the response to the query object.

A participant issuing a query can demand that the oracle responds at most N blocks after the query was accepted on chain. If no answer is received during this time span the query fee should be returned to the participant issuing the query.

As an usage example, let us assume that we are interested in monitoring a sensor, which can peak too high. Concretely the sensor is read by a (trusted) web service that responds to queries regarding its state using the blockchain mechanism. Thus the web service posts an **oracle register transaction** to the chain, registering a promise to answer queries regarding how many times the sensor peaked too high during, for instance, the last week.

As an example of how such oracles can be used in smart contracts, let us assume that two parties have agreed that if during a week the sensor reading is too high on, at least, say, 4 occasions then party 1 should pay party 2 a fixed amount of tokens as a compensation. Party 2, who is interested in getting paid, will post an **oracle query transaction** at least once a week. The web service that monitors the blockchain will see this query and post an **oracle response transaction** with the sensor data. In this way, the sensor data becomes part of the data on the blockchain. Both party 1 and party 2 can verify whether the value has been too high too often, and because they both trust the sensor readings, pay tokens accordingly. Such token payments can be enforced by the smart contract; a desired feature interaction between oracles and smart contracts that should be thoroughly tested.

Oracles have a specific lifetime, specified as the maximum permitted block height when registering the oracle. [After the blockchain has reached the specified maximum block height, the oracle will ignore further queries. However, note that the lifetime of an oracle can be extended using an **oracle extend transaction**.](#)

To pay for oracle operations, funds (tokens) must be transmitted to oracles from participants issuing oracle queries. In fact, in Aeternity oracles are accounts, which can spend surplus funds to other accounts.

3.4 | Naming system

Transferring tokens to a registered name, instead of a hard to remember public key, is a feature that is supported natively by Aeternity. For example, in a spend transaction the recipient can be given as a registered name. For that to work, a collection of 5 types of transactions are provided that resemble internet domain name (DNS) registration. With the **name service preclaim transaction** one can reserve a name, that can later be claimed (registered), without revealing what name one actually wants to claim. The **name service claim transaction** is then used to actually obtain the name, when available, and in a separate **name service update transaction** the name can be made to refer to something (for example an account). Additionally there is a **name service transfer transaction** to change the owner of a name, and a **name service revoke transaction** to free the name.

Names have a specific lifetime and there is an agreed minimum number of key-blocks before one can claim a name after a pre-claim. Similarly, there is a maximum number of blocks one can wait between a pre-claim and a claim in order to succeed with the claim. Obviously, registered names expire after a while, unless renewed in time with the update transaction.

For ensuring the correct operation of the naming system it is important to test combinations of basic operations such as e.g. transferring names, updating pointers, while spending to the named accounts, as these features heavily interact. Other potentially troublesome cases that need to be tested include revoking a name while using it, as well as spending to unregistered names, etc.

There are also feature interactions, which need to be tested, concerning the use of names in conjunction with oracles and contracts. The Aeternity naming system, for instance, allows to use a name as an account. Since oracles are also accounts, it should be possible to query an oracle by its name. Contracts are associated to accounts, but cannot be called by the account name, since one account can be associated to many different contracts.

3.5 | State channels

State channels were introduced to allow transactions and smart contracts between two blockchain participants (the initiator and the responder) to be executed off-chain to improve the scalability of the blockchain²⁵, while still recording the settlements and disagreements regarding such transactions and contracts on the chain, as well as managing the token (money) flow. The price to pay is that one needs to have a bit of trust in the other party in the state channel.

Off-chain communication can be done much faster and much cheaper than on-chain communication. There is, for instance, no need to wait until blocks are mined, and miners do not need to be paid. Moreover, moving communication off-chain which do not need the persistence guarantees provided by blockchain data can greatly improve the scalability of the blockchain.

To open a channel the connected parties should agree on how many tokens the channel should contain, how much of those the initiator and the responder to the channel put in, and what other data the initial state of the channel should contain. This agreement results in a mutually signed **channel create transaction** that is posted on chain. After that, the parties communicate off-chain.

In a friendly setting, for example a person renting a bike and being charged per kilometre, every now and then an off-chain state channel transaction is created in which both parties agree upon the new state. The bike rental firm could for instance send an update after each kilometre, asking the renter's app to sign this. At the end of the bike ride, the renter signs the total distance, as does the rental firm. This mutual settlement is packaged in a **channel close mutual** transaction and posted on chain. Funds in the channel are redistributed to the initiator and responder according to the agreed ratio.

If the renter wants to cycle more than originally anticipated, it may be necessary to put more funds into the channel. This is achieved by posting a **channel deposit transaction** to the chain, again mutually signed. Similarly, both parties can agree that some money is withdrawn from the channel using a **channel withdraw transaction**.

This all requires both parties to sign. But if one of the parties is unwilling to sign, or just disappears, then there is a possibility to post a **channel close solo transaction** to the chain. Such a transaction should include the hash of the latest state that both parties agreed upon. The funds in the channel are then divided in a way beneficial to the closer of the channel. However, if this happens as a deceit, then the other party can post a **channel settle transaction** that presents a later agreed state hash and different redistribution of funds.

4 | PROPERTY-BASED TESTING

Property-based testing (PBT) is a testing methodology which focuses on generating, automatically, test cases from a more abstract description of the behaviour of the system under test (the property). That is, property-based testing can be considered a form of model-based testing.

Property-based testing originates from the functional programming community¹⁰. Nowadays, there are different types of PBT tools available for many programming languages. In this article we use Quviq Erlang QuickCheck⁸ (henceforth simply EQC), which uses the Erlang programming language to specify system behaviour. EQC is a PBT tool extended to test stateful systems, and uses a state machine description to judge whether a sequence of operations has been correctly executed.

4.1 | Basic PBT

The three key basic ingredients of PBT are generators, properties and shrinking.

4.1.1 | Generators

A generator is capable of generating an infinite number of values of some data type, according to a probability distribution. EQC comes equipped with a library of standard generators, for example, `int()` is a generator for integer numbers, `choose(10, 100)` generates an integer number between 10 and 100, and the generator

```
frequency([99, choose(10, 100)], [1, int()])
```

denotes that 99% of the times, a value between 10 and 100 is generated, while 1% of the times an arbitrary integer is generated. A user of a PBT tool can also implement custom generators. It is fair to say that key to a successful application of the PBT methodology is the informed development of generators to ensure that the test process explores a significant part of the behaviour of the program under test. Different PBT tools use different techniques to ensure this, including more classical coverage based analysis. However, purely random testing turns out to have its advantages too; see ¹³ for an interesting and thorough comparison on the different methods of test case generation.

4.1.2 | Properties

The test property examines the result of running the system under test with the generated values as parameters, and judges if the result value (or launched exception) is correct given the input parameters. In a simple, but precise, case we can simply check whether the returned value is equal to the expected value, but properties might also be less precise and more involved, e.g., checking that a procedure generating random prime numbers really returns prime numbers.

Thus, given a set of generators for input data, and a test property, EQC generates a random instantiation of the variables, executes the system under test, and checks that the resulting Boolean property is true. This procedure is by default repeated 100 times. If for some instantiation the property returns false, or a runtime exception occurs, an error has been found and testing terminates with a counterexample.

4.1.3 | Shrinking

As a final ingredient in PBT implementations, there is an attempt to derive an easy-to-understand counterexample through a procedure called shrinking which systematically tries to simplify counterexamples, in order to ease the (manual) analysis required to discover the cause of the detected error.

For instance, suppose a bug has been found in a function sorting a list. Then the counterexample (or witness) for the bug is the input list that caused the error. Given such an initial counterexample, PBT tools will then systematically try to reduce the size of the counterexample by deriving new (“simpler”) lists starting with the original counterexample, and check that they too fail the test property. To simplify a list, for instance, EQC will try to remove elements from the list, and simplify the elements in the list. For example, the lists `[], [10], [10, 12], [10, 0]` are by default all “simpler” lists than `[10, 13, 12]`. As for generators, the default shrinking strategy can be modified.

4.2 | EQC State Machines

In most cases, a test case is not a simple call to a function, but rather a sequence of calls to operations or methods implementing an API. That is, a generator returns sequences of API calls, and the test property checks whether the execution of such a sequence of API calls is correct. To generate such test cases, and to judge whether the execution of a test case is correct, EQC provides a state machine library (`eqc_statem`). This library is used to program a state machine which is used during both the two phases of testing: (i) test case generation, and (ii) test case execution and test correct checking. In fact the EQC state machine acts as a model for the program under test.

An EQC state machine has a state which can be understood as the model state of the system under test. During test case generation, given the current model state, the library generates randomly a suitable next API command, and calculates the next model state. In this way it is natural to express constraints on the generated test cases such as, for instance, that we must have created at least one data structure before attempting to invoke operations on such data structures. Note that during test case

generation the model state is typically symbolic, as we cannot know how the system under test will respond to any stimuli. If, for instance, the system under test may choose nondeterministically a return value to some API call, and we want to save that return value in the model state (e.g., to check that some later API call returns the same value), the concrete value returned by the system under test cannot be stored in the model state (as it is not available during test case generation) but EQC rather stores a symbolic representation of the return value (a fresh variable).

Next, once a test case (of random length) has been fully generated in the manner described above, EQC proceeds to execute the calls in the test case. EQC checks after a call has completed, whether the result was the expected one given the current model state of the state machine, and computes a next model state. Note that during test case execution the actual values returned by the system under test in response to API calls are available, and are stored in the model state. Thus, for executing an API call, EQC takes care to first substitute actual values for the symbolic variables that may occur in generated API calls.

To use the state machine library a user must then

- specify the initial state of the state machine. In our example the model state will consist of a list containing the created sets; initially the value is the empty list `[]`.
- decide on a particular set of “commands” or types of API calls which should be invoked during testing. In our small example there are two commands: `newSet` and `add`. For each such command `cmd` a user of the state machine library should implement a number of “callback” Erlang functions³, which are explained below.

- `cmd_pre(State)` which implements a precondition on the state of the state machine acting as a constraint on when it is sensible to issue the command. For instance, when testing the set implementation, there are no constraints on when a new set can be created. However, a precondition for issuing an `add` command, is that a prior command in the test sequence must have created a set. In practice we check that the model state is not the empty list (as we store created sets in the model state).
- `cmd_args(State)` which returns as a list the arguments to the command. For instance, to complete a set “add” command the first parameter should be the set (typically stored in the model state), and the second argument could be a randomly generated integer. Our specification then becomes:

```
add_args(State) -> [oneof(State), int()].
```

where `oneof` is a EQC generator that picks, randomly, any generator in its list argument (note that a ground value is a generator for that ground value), and `int()` generates a random integer.

- `cmd(Arg1, ..., ArgN)` which is the implementation of the command. In our example, we must write the functions `newSet()` and `add(Set, Element)` to really create a set and add an element. Note that this function is called during test execution only, and thus has access to the real values returned by the implementation.
- `cmd_post(State, Args, Result)` which should examine the result (`Result`) of executing the command and judge, based on the model state and the command arguments, whether the result is correct. That is, the function implements the postcondition check for the command. In our example the only error is if the execution by the commands are aborted by exceptions; as this is checked by default we do not have to implement the postcondition functions for the set operations. Note that this function is called during test execution only, and thus has access to the real values returned by the implementation.
- `cmd_next(State, Result, Args)` which should compute the next (model) state, given the current model state, the command arguments, and the result of the executing the command. Note that since this function is called both during test case generation and test execution the function must be able to work with symbolic results and, possibly, symbolic command arguments. In the example, we implement

```
newSet(State, _Args, Result) -> [Result|State].
```

which adds the newly created set (which may be a symbolic variable) to the rest of the created sets stored in the model state.

³That is, functions which are defined by the state machine library user, and which are called from within the state machine library

4.3 | Test Coverage

Using PBT does not liberate us from the task of evaluating what we really have tested. It is only too easy to generate random program inputs, and because of the huge number of possible values, never get to explore the truly important inputs. As a motivating example, to test a method for sorting an array of integers it is surely important to test whether the method works correctly for the case when the array contains duplicated numbers. However, if we choose to test the sorting of many small arrays of integers, and moreover select the integers (in the array) from the whole range of integers, then **it is unlikely that we will ever test an array containing duplicated elements.**

In traditional testing we use traditional testing measures such as line coverage (or path coverage, etc.) to ensure that we test a system sufficiently. The line coverage measure, given a test suite, details which lines (e.g., percentage of lines) in the system under test were executed during the execution of the test suite, and so for our sorting example would likely warn us that we are not testing source code lines which cater to the case when elements are duplicated. The PBT tool EQC supports calculating such traditional coverage measures too.

However, if during a test session using EQC the e.g. line coverage is found to be poor, this in itself does not significantly aid testers on how to improve the testing. A problem has been found, but no solution identified. In testing Aeternity, we have opted to measure test efficacy through the use of a so called “feature coverage” measure²⁶. EQC permits to annotate generated commands with a number of “features” that they test for, and for a test suite report the percentage of test cases that test for that feature. In this way we can, for instance, label arrays with duplicated numbers as “duplicated” and those without as “non_duplicated”, and an absence of testing for the “duplicated” feature becomes immediately obvious. To differentiate between blockchain features and these features, we will use the word “measurements” when we refer to the latter in the rest of the paper.

5 | TESTING THE VALIDITY OF TRANSACTIONS

The EQC model focuses on testing the core of the system, i.e., the blockchain transaction engine comprising around 15,000 lines of Erlang code. The EQC model does not test other system concerns such as mining, synchronisation, transaction mempools, the HTTP interface and such. The EQC model comprises around 2550 LOC, compared to over 35,000 LOC in unit tests (which do test other system concerns).

For testing the transaction core we do not use the “real” distributed Aeternity blockchain (at block height 299194 when writing this phrase) but rather a local, non-distributed, and initially empty chain. Testing can begin from any block height, permitting to check different consensus rule sets (e.g., differing contract gas minimum transaction costs and fees, and contract language implementations).

In the EQC model, the logic for each transaction is described separately as a set of Erlang functions. In EQC terminology, the model code for a transaction is known as a “command”. In the model we describe in this article, the model tests 18 different Aeternity transactions, comprising basic fund transfer operations (spend), oracles, state channels, a naming service, and smart contracts. The spend transaction, for example, is described in around 70 lines of code.

Mining is an additional EQC command in the model. This command simulates a mining operation by updating the blockchain state trees directly and incrementing the blockchain height, without using e.g. transaction pools or performing actual (proof-of-work) mining, thus permitting to greatly increase test execution speed. By implementing mining as an EQC command, we can obtain a random interleaving of transactions with mining operations, just as can be expected in real blockchain operations. Moreover, long running smart contracts whose operations frequently depend on blockchain heights (as an implicit time parameter), can be tested under different mining behaviours. Having explicit control of mining in the EQC model is vitally important, as the blockchain height is a central operation in many transactions, serving as a time parameter. For example, the time-to-live (TTL) parameters for oracles, oracle queries, and the name registration procedure, are expressed in terms of blockchain height.

Table 1 enumerates the commands of the EQC model, including the commands testing the 18 Aeternity transactions. The so called “environmental commands” sets up the blockchain for testing, e.g., defining the initial blockchain height, and adding a set of already known users to the blockchain. The choice of a good test setup is discussed further in Sect. 7.

In the remainder, we illustrate the approach by explaining some carefully chosen parts of the EQC model. The full EQC model for testing Aeternity described in this article is available as open source at <https://github.com/aeternity/aeternity-eqc>.

<i>Environmental commands</i>
init, newkey
<i>Blockchain mining</i>
mine, multi_mine
<i>Spend transaction</i>
spend
<i>Oracle transactions</i>
register_oracle, extend_oracle, query_oracle, response_oracle
<i>State channel transactions</i>
channel_create, channel_deposit, channel_withdraw, channel_close_mutual, channel_close_solo, channel_settle
<i>Naming system transactions</i>
ns_preclaim, ns_claim, ns_update, ns_revoke, ns_transfer
<i>Smart contract transactions</i>
contract_create, contract_call

TABLE 1 Model commands

5.1 | Modelling the spend transaction

To illustrate the design of the EQC model for transactions, we begin by explaining the EQC model for the spend transaction.

5.1.1 | Generating data for the spend transaction

The spend transaction needs a number of arguments; in the EQC model such command arguments are specified in the `spend_args` function which returns a data generator for a list of arguments. A first attempt at generating arguments for the spend transaction is shown below.

```
spend_args(S) ->
[ #{sender_id => pubkey(32),
  receiver_id => pubkey(32),
  amount => int(),
  fee => int(),
  nonce => int()} ].
```

The above function returns a list with one argument, which is an Erlang map (a functional key-value store) describing the transaction. The `sender_id` is the Aeternity actor paying for the transaction. The `receiver_id` receives the amount specified in the transaction from the sender (note that a receiver may be previously unknown to the Aeternity blockchain). In the above transaction generator, the sender and receiver of a transaction take a random public key (which, obviously, is not very effective if we want to generate valid transactions). The spend amount, fee and nonce are random integers. In Aeternity, the nonce is a strictly increasing counter associated to an account in order to prevent reply attacks. Thus, obviously, selecting just random nonces is a very bad testing strategy. Moreover, negative amounts are disallowed in the spend transaction, and result in an error. In conclusion, both positive and negative tests are generated.

For the above reason, although this generator creates both valid and invalid spend transactions, most of the generated tests will not be “interesting”, i.e., almost all tests are likely to be executed by the same code fragment which rejects invalid transactions.

To find out how bad our test models behave, we instrument generated test cases with measurement annotations in the style of²⁶. That is, we annotate the generated spend transactions with labels indicating what they test, e.g., whether a spend transaction is valid or invalid, and the reasons for rejection. Such annotations are collected during test execution, and afterwards displayed to the tester. To declare that the spend transaction has a number of features we must program a new callback function, `spend_features`, which returns a list with the features tested by a particular call with the generated transaction Tx:

```
spend_features(S, [Tx], _Result) ->
```

```
[ {spend, ok} || spend_valid(S, [Tx]) ] ++
[ {spend, {error, invalid_tx}} || is_invalid_tx(Tx,S) ] ++
[ {spend, {error, sender_account_unknown}} || not(is_sender_known(Tx,S)) ] ++
...
```

The function checks that the generated transaction Tx is valid given the current testing state S (`spend_valid(S, [Tx])`), and otherwise returns a list of detailed detected errors. For instance, the function tests verifies that the specified sender account is present in the testing state, if not an error `{error, sender_account_unknown}` is signalled. Note also that a single transaction can fail for multiple reasons – the error “features” are cumulative since a list is used.

The diagnostic output of a test run is whether an error was detected, and a summary of the percentage with which each command (transaction) was generated. For these tests only the spend transaction is enabled, together with the needed machinery for testing the blockchain: creating new public keys, mining blocks, and starting the blockchain. Next, for each command, the percentage of features tested by the command is printed.

In our case, where the `int()` generator has 50% possibility to generate a negative, and therefore, invalid value for 3 of the fields, we obtain the following statistics after running 10,000 tests (156637 spend transactions):

```
47.5% {txs_eqc,spend,4}
24.5% {txs_eqc,newkey,2}
16.6% {txs_eqc,multi_mine,2}
 5.7% {txs_eqc,mine,1}
 5.7% {txs_eqc,init,1}
...

spend
88.629% {error, invalid_tx}
...
0.000% ok
```

Disappointingly, none of the generated spend transactions were valid, due to e.g. the presence of negative numbers in the transaction data (`invalid_tx`) in around 89% of the spend transactions generated.

To improve this model – not even one correct transaction is tested – one possibility could be to use, for instance, fuzzing techniques, e.g.,^{27,28}, which would usually use machine learning techniques to try to avoid generating an excessive number of syntactically incorrect transactions. In this approach, instead, model design is essentially a programming task. Thus, we want to provide the model programmer with the right tools for debugging such model problems, e.g., showing what is actually tested through the use of the measurements mechanism. Given this insight, the model programmer will then steer the distribution of valid transactions. Steering manually the test distribution of a complex system such as the transaction core of a blockchain is admittedly a hard problem, but so is attempting to derive automatically a good distribution (see e.g.²⁹ for a recent work in automatic generator derivation). Concretely, we need to “almost always” use correct amounts in the spend transaction, almost always use existing accounts, and almost always use the current account nonce. For instance, still a bit naively, we can use the generator

```
fee => frequency([{99, choose(10, 100)}, {1, int()}])
```

for the amount. 99 out of 100 times chooses a value between 10 and 100 as the amount, and 1 out of a 100 an arbitrary integer is chosen, including negative amounts which result in syntactically invalid spend transactions.

In order to generate spend transactions that can be successfully applied to the chain, we need to use, e.g., valid public keys. For this we use the notion of a stateful EQC model, as which keys are valid depends on which transactions were executed before the spend transaction was created.

5.1.2 | EQC state machine model for transactions

The sequence of commands that EQC generates can use and modify a test state. For the spend transaction we need to keep track of accounts in the blockchain. The test state is a map which has a key accounts with the account records as value, a set of known public and private key pairs, the current height of the blockchain, and a list of payed fees:

```
#{accounts => [FirstAccount], height => InitialHeight, keys => #{...}, fees => []}.
```

where FirstAccount is a known account at the start of each test, which is configured in the blockchain, and which moreover has enough funds to permit executing a number of transactions. This account has the Erlang type

```
-record(account, {key, amount, nonce}).
```

That is, a composite structure (an Erlang record) consisting of a public key identifying the account, the amount remaining in the account, and the current nonce.

Later, when describing the support for testing other blockchain features, e.g., contracts and names, the test state will be extended to support testing these additional features.

Now each generated command can use the state. We examine below how the spend transaction test code is modified to take into account the test state. First, to increase the possibility of valid transactions we define a generator of accounts `gen_account(PNew, PExisting, S)` that with weight PNew generates a new public key, and with weight PExisting reuses a public key already associated with an account in the test state *S*. It returns a tuple {Tag, PublicKey} where the tag indicates whether the key is new or was already present in the test state, permitting us to observe with what frequency new or existing accounts are used as arguments to transactions in the test run. Next, we can generate the nonce parameter using a generator `gen_nonce(SenderAccount)` that with a high probability generates a correct nonce (i.e., identical to the nonce field of the account parameter), and with a low probability an incorrect nonce. Next the spend amount is generated to likely be less than the amount in the sender account using the generator:

```
gen_spend_amount(#account{ amount = X }) when X == 0 ->
    choose(0, 10000000);
gen_spend_amount(#account{ amount = X }) ->
    weighted_default({49, round(X / 5)}, {1, choose(0, 10000000)}).
```

Finally, the fee is initialised to a likely correct value, taking into account the minimum gas price at the current height of the blockchain using the generator `gen_fee(S)`; recall that the height of the blockchain is recorded in the test state too.

The modified generator for spend arguments then becomes:

```
spend_args(#{height := Height}, S) ->
    ?LET(Sender={SenderTag, SenderAcc}, gen_account(1,49,S),
        ?LET(Receiver={RecvTag, ReceiverAcc}, gen_account(2,1,S),
            ?LET(Nonce, gen_nonce(SenderAcc),
                [
                    Height, Sender, Receiver,
                    #{sender_id => SenderAcc#account.key,
                      receiver_id => ReceiverAcc#account.key,
                      amount => gen_spend_amount(SenderAcc),
                      fee => gen_fee(S),
                      nonce => Nonce}
                ]))).
```

Note that for convenience we supply the current blockchain height, and the sender and receiver accounts and tags, as arguments to the spend function. With these changes, we get a test distribution as follows:

```
spend
91.18% ok
6.44% {error,too_low_fee}
1.34% {error,account_nonce_too_low}
```

Most of the spend transactions are now valid.

This summarises the basic steps needed to generate meaningful blockchain transactions. For each transaction that we want to test, we create a API call generator (named `_args`). We use random generators with carefully chosen distributions to generate good test data. Sometimes, the data needs to be constructed from the state we assume the test to be in.

To get a complete EQC model we need to describe how transactions are actually executed on the blockchain, i.e., implementing the spend function, updating the state by implementing the `spend_next` function, and finally we have to judge whether the blockchain correctly executed the transaction.

5.1.3 | Executing the spend transaction

A generated spend transaction is executed by the `spend` function:

```
spend(Height, _Sender, _Receiver, Tx) ->
  apply_transaction(Height, aec_spend_tx, Tx).
```

The concrete details for how the spend transaction is added to the simulated blockchain is omitted in this article; see https://github.com/aeternity/aeternity-eqc/blob/master/aecore_eqc/txs_eqc.erl for more details.

In the simulated blockchain we use for testing, the whole state of the blockchain is represented as a functional Merkle tree structure in memory, which as a side effect is stored in the Erlang process dictionary. Recall that although principally a functional programming language, Erlang has a process local store called the process dictionary to which a process can write and read using the `put` and `get` functions. Thus we can save the whole blockchain before executing a transaction, and analyse the differences compared to the blocking resulting from executing the transaction.

5.1.4 | Valid transactions

The execution of a (spend) transaction should update the state only if the transaction is valid. Similarly, to judge whether the execution of a transaction by Aeternity is correct, we need to check that valid transactions are accepted by the blockchain, and conversely, that invalid transactions are rejected.

As the validity check is needed in several places, we define a function `spend_valid` that checks whether a spend transaction should be accepted:

```
spend_valid(S, [Height, {_, Sender}, _Receiver, Tx]) ->
  is_account(S, Sender)
  andalso correct_nonce(S, Sender, Tx)
  andalso check_balance(S, Sender, maps:get(amount, Tx) + maps:get(fee, Tx))
  andalso valid_fee(S, Tx).
```

The function checks that the sender is a known account, that the nonce is correct for that account, that the account has enough balance to pay for the transaction amount and the proposed fee, and that the proposed fee is sufficient according to the current consensus (at the current block height).

5.1.5 | Updating the test state

The function `spend_next` returns a new test state, and receives as argument the current test state, the value returned when the generated spend transaction is called (`_Value`), and the arguments of a generated spend transaction. Recall that such `_next` functions are called both during test case generation and test case execution, and thus the returned value may in fact be a symbolic variable.

```
spend_next(S, _Value, [_, {_, Sender}, {_, Receiver}, Tx] = Args) ->
  case spend_valid(S, Args) of
    false -> S;
    true ->
      #{ amount := Amount, fee := Fee,
        receiver := Receiver } = Tx,
      S1 = credit(Receiver, Amount, S),
      S2 = credit(Sender, -(Amount + Fee), S1),
      S3 = bump_nonce(Sender, S2),
      reserve_fee(Fee, S3)
  end.
```


In the case the spend transaction is a valid transaction (checked by `spend_valid`), the `credit` function returns a new state where the account balance argument has been increased with amount argument; note that the function possibly creates a new account in the test state as the receiver may not have existed prior to the execution of the spend transaction. In the first call to `credit` the account balance of the receiving account is increased with the transaction amount, in the second call the account balance of the spending account is decreased with the sum of the amount and the transaction fee. The `reserve_fee` function adds the new fee to the fees recorded in the state and returns a new test state, and similarly the `bump_nonce` function increments the nonce of its argument account. Note that the fees in the state are used to check, upon completion of running a test case, that the blockchain accounts has not lost money except the fees which have been diverted to blockchain miners.

5.1.6 | Verifying the execution of the spend transaction

Every transaction is checked by the same function `postcondition_common` which receives as arguments the current test state, a symbolic representation of the transaction call, and the result:

```
postcondition_common(S, {call, ?MODULE, Fun, Args}, Res) ->
  IsValidTransaction = valid_common(Fun, S, Args),
  case Res of
    {error, _} -> not(IsValidTransaction);
    ok         -> IsValidTransaction
  end.
```

The `postcondition_common` function simply checks that if the transaction outcome prediction (`valid_common`) coincides with the result of actually executing the transaction on Aeternity (i.e., an error `{error, _}` or `ok`). The `valid_common` function calls the specific valid function for a given transaction type, to predict the transaction outcome:

```
valid_common(spend, S, Args) -> spend_valid(S, Args);
...
```

It may surprise the reader that we do not check that e.g. the real balance of the blockchain accounts of the sender and receiver have been updated correctly by the execution of a spend transaction. We could certainly do this, by implementing a transaction type specific `spend_post` function, and analysing the differences between the blockchains before and after execution the spend transaction (recall that we save the blockchains data structures in the Erlang process dictionary). However, we instead rely on later transactions in the generated test case to detect such errors due to differences between believed account balances (the test state) and real account balances. That is, believed valid transaction will be rejected by Aeternity because not enough funds are available in the sender account to pay for the amount sent and the mining fees (in the case of a spend transaction).

In the general case, whether to check the result of each model command more thoroughly, or rely on later commands to detect the error is a modelling trade-off. Detecting errors as early as possible is clearly beneficial, however, it does require adding extra code to the model to detect such errors. In practise, it may be that the cost of adding such code outweighs the benefits of early detection, especially as EQC is normally rather good at producing small test case counterexamples⁴, thus limiting the distance (in the counterexample test case) between the source of an error and its detection.

5.2 | Modelling the oracle transactions

To illustrate how the combination of different blockchain features is modelled cleanly, we next show how the Aeternity oracle feature is introduced in the EQC specification.

In short, Aeternity oracles provide a mechanism for importing data from outside the blockchain into the Aeternity blockchain. Oracles can be used, for instance, to provide financial data, which can be utilised by smart contracts. In Aeternity, support for oracles are provided by three different transactions. To provide support for oracles in the EQC model requires around 250 lines of new code, but also requires us to modify some existing functions.

Recall that oracles are created using the **oracle register transaction**, by some blockchain participant for the purpose of answering queries from (other) participants. Other blockchain participants may query the created oracle for information using the **oracle query transaction**, supplying funds to pay for the result; this creates an oracle query object on the blockchain. The

⁴through the shrinking feature, as explained in QuickCheck references

oracle responds to such queries using the **oracle response transaction**, which adds the response to the query object. In the following we will illustrate how to test the **oracle register transaction**. The first step is to define a generator for the transaction:

```
register_oracle_args(S = #{height := Height}) ->
  ?LET(Sender = {SenderTag, Sender}, gen_new_oracle_account(S),
    [Height, Sender,
      #{account_id => aaser_id:create(account, Sender#account.key),
        query_fee => gen_query_fee(),
        fee => gen_fee(S),
        nonce => gen_nonce(Sender),
        oracle_ttl => {delta, frequency([9, 1001], {1, elements([0, 1, 2][])})})}).
```

where there is a specific fee for posting a query to the oracle and, because oracles have a specific lifetime, a time-to-live (TTL) value is specified. Note that, for the sake of clarity, we have omitted some fields of this generator. Given the state of the EQC model and a generated transaction for registering an oracle, the following function checks whether this transaction should be accepted.

```
register_oracle_valid(S, [Height, {_, Sender}, Tx]) ->
  is_account(S, Sender)
  andalso correct_nonce(S, Sender, Tx)
  andalso check_balance(S, Sender, maps:get(fee, Tx))
  andalso valid_fee(S, Tx)
  andalso (not is_oracle(S, Sender) orelse oracle_ttl(S, Sender) < Height).
```

In this case, the sender pubkey should be one of an existing account that has the correct nonce and enough balance to pay for the fee. In addition, the sender should not have registered an oracle or, if it has registered an oracle, the oracle has expired.

Then, the computation of the next state is specified as follows:

```
register_oracle_next(S, _Value, [Height, _, Tx] = Args) ->
  case register_oracle_valid(S, Args) of
    false -> S;
    true ->
      #{fee := Fee, queryFee := QFee, oracle_ttl := {delta, D} = Tx,
        Oracle = #oracle{id = Sender, qfee = QFee,
          oracle_ttl = D + Height},
        reserve_fee(Fee,
          bump_and_charge(Sender, Fee,
            add(oracles, Oracle,
              remove(oracles, Sender, #oracle.id, S))))).
  end.
```

If the transaction is not valid, the state remains unchanged, otherwise, the balance and the nonce of the sender account are updated, the oracle that has been registered is added to the list of oracles kept in the state and the sender removed from the list of accounts in the state. Lastly, the fee for the miner of the block is reserved.

In a similar way, we add the other oracle transactions to the EQC model.

5.3 | Testing Feature Interactions

Adding new features to a blockchain (oracles, named accounts, contracts, etc.) may also require changes to existing model code. This is unavoidable, as the introduction of new features in the Aeternity code extends and interferes with existing functionality. Sometimes such interactions are expected, e.g., smart contract code has to be able to distribute funds kept in the contract account to normal accounts, and sometimes the results are perhaps not well thought out, e.g., what happens if we try to send funds to a smart contract directly using the spend transaction? Does the blockchain permit this? Does the smart contract code take this possibility into account?

Clearly, we need to modify the EQC model to generate test cases which test possibly feature interactions, i.e., in the same test case mixing calls to transactions belonging to different features (e.g., naming, oracles and spends), and moreover the arguments in these transactions should be intermixed too. Thus, we should generate test cases such as registering a name for an account, spending to the account, and later using the account name to pay for an oracle query transaction.

Concretely, we have to generalise pre-existing transactions (e.g., spend) to permit different types of accounts. Then, we should inspect the test cases generated to analyse whether we have done sufficient feature interaction testing, or whether we should modify generators, and the distribution of transactions executed, to do better testing. We illustrate this methodology in the scenario where we have already implemented the spend transaction in EQC, and proceed to explain the changes necessary to support spending to oracles, and to named accounts (for an introduction of named accounts see Sect. 3.4).

First we have to modify the generator of spend transactions to permit spending to oracles and named accounts, this is done in `spend_args` where we permit the recipient to be, with a high probability, any type of account, and with a lesser probability an oracle account or a named account.

```
{TypeOfReceiver, Receiver} =
  frequency([10, {account, gen_account(2,1,S)}},
            {2, {oracle, gen_oracle_account(S)}},
            {2, {name, gen_named_account(S)}}])
```

The `oracle_gen_account` generator returns an oracle account with a high probability (if such accounts have been created), or otherwise a fake account identifier (for negative testing). Similarly `gen_named_account` generates a named account with a high probability (if such exists), and otherwise a fake name.

Next we have modified the `spend_valid` function to accept also named accounts by modifying the condition on the receiver:

```
{TypeOfReceiver, ReceiverName} = Receiver,
case TypeOfReceiver of
  name -> is_valid_name_account(S, ReceiverName, Height);
  ...
end
```

As we have seen, these updates are rather easy to make, consisting of few lines of code added to the spend transaction and the valid function. By adding this code, EQC would generate tests for spending to named accounts whose registration have not yet finalised (are in pre-claim state – see Sect. 3.4 – thus failing), to names that are claimed in the same block, or claimed several blocks before, names that change owner, oracles, and names that are revoked or have expired. None of these tests has to be explicitly written, they just follow from different (random) ways of putting the transactions together. Note that we can for instance reduce the probability of generated names for accounts that are in pre-claim state, and thus increasing the probability of generating successful spend transactions, by modifying the `gen_named_account(S)` generator. To judge whether tailoring with the generator is necessary we need to measure the generated tests.

To do this we modify the measurement function for spends, to report when named accounts are used. In addition, we let the function label spends when the sender and recipient are identical:

```
spend_features(S, [_Height, {_, Sender}, {Tag, Receiver}, Tx] = Args, Res) ->
  Correct = spend_valid(S, Args),
  [{correct, if Correct -> spend; true -> false end}] ++
  [ {spend_to, self} || Sender == Receiver andalso Correct] ++
  [ {spend_to, recipient_id(Receiver)} || Sender /= Receiver andalso Correct] ++
  [ {spend, Res}].
```

The function `recipient_id` returns the type of receiver (name, normal account, ...). From the 91% of successful spend transactions, the distribution for the specific measurements of spending to are:

```
spend_to
53.00% account
36.67% self
 8.52% oracle
 1.47% contract
 0.35% name
```

Since names and oracles are picked with the same probability when the receiver is generated, we might have expected a higher ratio of spending to a name. The reason for the low number is that creating a valid name is a rather involved procedure, involving a number of transactions in sequence, and moreover, registered names have a lifetime which can be exceeded during testing. To increase the probability of generating a spend transaction with a valid name there are two possibilities. We can (i) increase the number of spend transactions relative to other transactions, or (ii) we can increase the possibility that a spend transaction uses a named account as the recipient.

To realise option (i) we note that in EQC one can specify a weight for each command (transaction). This is specified as the callback function

```
weight(TestState, CommandName)
```

which should return a weight (a non-negative integer) corresponding to how desirable it is to generate the command (e.g., a spend transaction) in that test state. Concretely EQC chooses as the probability for generating a command in a state the ratio between its weight and the sum of weights of all commands. We could simply increase the weight of the spend (transaction) command. However, this will unlikely produce more spend transactions to a named recipient, since prioritising send commands may well deprioritise the transactions which registers names, which are needed for a successful spend to a registered name. Instead, we can prioritise the spend transaction when there is a registered name to spend to:

```
weight(State, spend) ->
  case has_registered_name(State) of
    true -> 20;
    false -> 5
  end; ...
```

Using the `weight` mechanism in this manner is commonly used to focus test case generation towards progressing commands.

To realise the second option, i.e., prioritising using named arguments in spend transactions, we modify the `spend_args` to prioritise named recipients compared to other recipients when there is a registered name to spend to:

```
WeightNamed =
  case has_registered_name(State) of
    true -> 10;
    false -> 2
  end,
{TypeOfReceiver, Receiver} =
  frequency([
    {10, {account, gen_account(2,1,S)}},
    {2, {oracle, gen_oracle_account(S)}},
    {WeightNamed, {name, gen_named_account(S)}}])
```

With this second change, the following distribution of spend recipients is reported:

```
spend_to
46.5% account
38.4% self
8.8% oracle
6.3% name
```

At least both valid name and oracle recipients are being tested now, although we should definitely test less spends where the sender and recipient are the same (`self`).

5.4 | Correct Test Executions

EQC judges the whether the Aeternity blockchain correctly executed the generated sequence of transactions on two criteria: (i) transactions are correctly accepted/rejected according to the test model oracle (see previous discussion in Sect. 5.1.4), and (ii) the total sum of electronic money in the Aeternity blockchain is equal to the sum of money before running the transactions of

the test case, minus the fees accrued from the transactions (which go to the blockchain miners). That is, the blockchain does not leak money.

The fees for normal transactions are randomly generated by the `gen_fee(S)` generator. For contract calls, additionally the gas used must be determined before issuing the call. For arbitrary contracts, this is impossible. However, the EQC model we use to check the correct operation of the blockchain issues only contract transactions whose gas usage is already known.

6 | RESULTS

To evaluate the result of using property-based testing to test the Aeternity blockchain, we consider the following three questions:

- What errors were found in the Aeternity blockchain using the EQC model that had not already been found using traditional testing?
- Are the tests generated by EQC for testing feature interactions generated with sufficiently high probability?
- What are the metrics of the EQC model for generating such tests and evaluating whether the test pass?

These issues are discussed in Sect. 6.1, Sect. 6.2, and Sect. 6.2, respectively.

6.1 | Finding issues in the Aeternity blockchain

Property-based testing was applied while the Aeternity blockchain was being developed. We found a number of issues when testing the validity of transactions. By talking to the developers, we found a few places where the documentation deviated from what was actually implemented. For example, it was documented that an oracle could be queried by its name, but the implementation failed to do so. A *TO-DO* in the code revealed that the documentation was ahead of reality. We also learned while testing that spending to a contract can succeed, even if the contract does not exist. The key used for the contract is then used as an arbitrary account key.

We also found a number of smaller issues when testing a major code refactoring. At first, mainly differences in the order in which errors are reported. If both the fee is too low and the nonce is incorrect, there is some freedom in which error to report first. Since we are mainly interested that no invalid transactions are added to the chain, we considered all errors equal.

One of the errors we spotted in refactored code was by the following minimised test case: create an account with some tokens, and spend all tokens to oneself with fee 1. The production implementation returned an error on this: `not_enough_funds`. The refactored implementation happily accepted this transaction and subtracted 1 from the balance to pay for the fee.

Another issue found was to do with state channels. When one opens a state channel, then the initial deposit is subtracted from the initiator and responder balances. But if the initiator and responder were the same account, then we had a case where only one of the deductions actually happened, whereas the full amount ended up in the channel. By closing the channel, the never deducted tokens could end up as surplus in the account balance.

In summary, the majority of the errors found revealed minor discrepancies between the behaviour of the blockchain and its documented behaviour. Either, (i) the blockchain signalled an error when the documentation did not, or (ii) the blockchain did not signal an error when documentation stated that it should, or (iii) the error message indicated by the blockchain was different from the documented error message.

However, there were a couple of errors detected which were dangerous (i.e., possibly exploitable), concerning features that had not been sufficiently tested previously.

In retrospect, it is easy to see that some unit tests were missing to detect such issues. But the point is that one does not know whether a unit test is missing. With the EQC specification we do not actively look for those tests, they just happen to be generated.

6.2 | Probability of generating feature interactions tests

In our experiments, we could run around 20 tests per second on a developer laptop (MacBook Pro). The length of the tests varies, but on average there are around 250 transactions per second. Even though mining is simulated, we observed that incrementing the height is one of the more time consuming operations to perform. Some tests build a chain of 50,000 blocks or more.

We tuned our generated data distribution such that we generate valid transactions 60% of the time. Depending on whether different distribution we tried, we can make each individual transaction valid more than 50% of the time, but this does not hold for each run. The distribution of the different commands over a larger number of tests can be seen in Table 2. Here we see that, for example, `ns_claim` is only 36.963% of the time valid, due to the fact that we only use a limited set of names, we often get the error `name_already_taken`. But we want to use a limited set of names to increase the possibility to generate negative tests claiming the same name twice.

Also visible in this table is that we provoke 138 different error messages, which means we run at least that number of different negative tests.

%	operation	successful	nr error messages
0.8155%	init		
3.0061%	mine		
1.0781%	multi_mine		
9.1109%	spend	89.202%	7
6.5896%	register_oracle	62.834%	6
6.4409%	query_oracle	74.653%	8
2.9699%	extend_oracle	86.427%	6
4.5561%	response_oracle	54.877%	4
7.6635%	channel_create	84.145%	7
4.1737%	channel_deposit	27.866%	9
2.7085%	channel_withdraw	4.566%	10
4.1621%	channel_close_mutual	64.963%	7
4.1621%	channel_close_solo	63.759%	9
4.1480%	channel_settle	37.914%	8
3.8908%	ns_preclaim	90.623%	5
7.0668%	ns_claim	36.963%	10
6.4399%	ns_update	51.625%	9
2.3708%	ns_revoke	31.253%	8
3.7089%	ns_transfer	18.488%	9
8.6426%	contract_create	53.552%	8
4.4652%	contract_call	74.125%	8
1.7724%	newkey		

TABLE 2 Operation distribution for 10,000 generated tests

6.3 | Metrics of the EQC model

The EQC specification comprises around 2550 lines of code (LOC). The specification is mainly organised around the concept of Aeternity transactions, which are implemented as EQC commands. The model code for a transaction is implemented by, roughly, 7 distinct functions (`cmd_pre`, `cmd_next`, `cmd_post`, etc.)⁵. In total there are 18 different transactions generated by the model. Moreover, there are 4 EQC commands for maintaining the blockchain: `init` – initialises the blockchain, `newkey` – generates new key pairs, and `mine` and `multi_mine` which implement different blockchain mining strategies.

Functionality	LOC
code for implementing commands	1636
common data generators	260
model state update and query functions	176
helper functions (e.g., for checking when a nonce is correct given the model state, etc.)	131
correctness property definition	90
weight function for determining the probabilistic distribution of different classes of commands	89
support code for handling contracts (e.g., contract compilation)	70
common pre/post-condition code	43
record declaration defining the model state	35
configuration details (e.g., defining blockchain versions and protocols, virtual machines, etc.)	33

TABLE 3 Code Sections

Apart from these transaction specific functions, the model contains the code sections with, roughly, the following lines of code as shown in Table 3. On average, then, the model code for each transaction (in the following we count mining operations, etc., as transactions too) comprises around 74 lines of code.

As blockchains in part compete with regards to their feature set, e.g. supported transactions and expressive contract languages, etc., an important aspect of model maintainability is how easy it is to add new transactions types (new features) to a finished model. In Aeternity, a feature (e.g., contract naming) is implemented as a set of new transactions (e.g., for naming, 5 such transactions). In our model, an EQC command implements each transaction type. As we have seen, on average, each command requires around 75 lines of code. However, supporting a feature in the EQC model also requires changes to other code sections. Most likely the model state has to be modified, new helper functions have to be written, the command weight distribution has to be modified, and new generator functions are needed. All in all, using a very conservative measure, by dividing the total number of model code lines with the number of commands a measure of 123 lines per code for each new command is obtained.

Regarding readability, the way in which transactions have been mapped to EQC commands provides a natural structuring mechanism for the blockchain model. Code to implement a feature is mostly located in the corresponding command(s), although, unavoidably, features do interact, leading the code for some commands (implementing a feature A) to explicitly mention some other feature (feature B). This occurs as we have seen, for instance, for the spending feature, where the target of a spend (where the electronic money is sent) can be types of accounts related to other features (names, oracles, smart contracts, etc.). A formal study on the topic of test code readability, in the style of³⁰, is left for future work.

7 | RECOMMENDATIONS FOR TESTING BLOCKCHAINS USING PBT

In this section, we reflect on the lessons learnt from this work with the goal of easing the application of PBT to test blockchain platforms. In fact, many of the lessons learned apply not only to blockchain platforms, but can also be applied to the task of modelling and testing other complex distributed systems whose functionality can be separated into different features or parts.

As explained in Sect. 4.2, the first two steps in constructing a PBT stateful model are: (i) to choose the initial state of state machine, and (ii) to identify the commands which will form the basic building blocks of a test case.

Choosing an initial state.

The initial state provides the initial testing environment. In our case this comprises the initial block height of the blockchain together with an initial user which must pay for the first transaction. An alternative is to add substantially more users (keypairs) to the initial environment, which varying account balances. Choosing the second alternative has the consequence that test cases become less self-contained as they rely on a more complicated initial environment which must be understood to understand the test case, but on the other hand, failing test cases become shorter and can possibly focus more clearly on an identified problem. Moreover, in general a more complex initial environment cannot be statically chosen, but should be “generated” during testing too. In general it is normally highly advisable to use a small testing environment, as we precisely would likely to test for incorrect “feature interactions”, and the probability of such interactions will only increase if the number of e.g. accounts is small.

³⁰The exact number varies, as functions have default implementations which may be adequate

Identifying commands.

For testing the validity of blockchain transactions, an obvious choice is to select the different transaction types as commands. In the case of testing the core Aeternity engine, examples of the functions that model the spend transaction are shown and explained in Sect. 5.1. Another example for one of the oracle transactions is given in Sect. 5.2.

In addition, we also included other commands which affect the environment in which the transactions execute. For Aeternity, this included two commands: one for mining transactions (increasing block height is similar to the advance of time), and another command to add a new user to the blockchain (adding a new pair of public and private keys).

Separating the model into different features.

We identified a number of separate “features” of the blockchain (e.g., oracles, name service) which extend the basic functionality of the blockchain. The PBT model reflects this separation of concerns by letting the model state be the union of the basic blockchain state and the separate features states. For example, in Aeternity, the basic state includes the current height of the blockchain and one user account. In addition, the state is augmented with Erlang records which store the information necessary for testing the different features of the blockchain. For example, there is one record called `preclaim` to store the names that have been preclaimed.

Moreover, the commands similarly either concern the basic functionality of the blockchain, e.g. the spend transaction and blockchain mining, or form a subgroup of transactions for a particular feature.

Identifying valid transaction leads to uniform computation of next model states and postcondition checks.

All basic transactions in the Aeternity blockchain behave the same: first the transaction is checked to determine if it is valid, i.e. syntactically valid and with semantically valid parameters, e.g., identifying existing accounts, etc. If a transaction is not valid, it is rejected by the blockchain. In the model we represent such validity checks explicitly by defining a valid function for each command, e.g., `spend_valid` checks that the spend parameters are valid given the current test state. Then, the postcondition check common to all transaction commands simply checks that the transaction result (ok or error) coincides with the prediction by the valid function. Moreover, the validity function is reused in the computation of the next state. If the transaction is not valid, the state does not change. Structuring the commands in this way avoids duplicating the same validity checking code in two places, and, in our experience, is applicable for modelling other concurrent/distributed systems too.

Checking test coverage using measurements.

Testing a complex system such as a blockchain is difficult. It is only too easy to generate, randomly, long sequences of commands where most of the later commands in the sequence are not valid transactions. This is a common difficulty for testing all kinds of complex transaction based systems. In this work we combat this difficulty by measuring accurately using the QuickCheck “features” mechanism what commands are generated, and whether the transactions they represent are valid at the time of execution. For example, that an **oracle response transaction** really refers to a defined oracle, and so on. Moreover, using the same mechanism we check that transaction parameters vary over all permitted types, e.g., are both basic account identifiers and symbolically registered names, etc.

Generating varied commands ensuring a good test coverage.

Using the measurement mechanism discussed in the previous item we can accurately judge whether a set of test cases test a system exhaustively. If not, the generators used to generate test commands must be changed. In this work this is an iterative manual process where e.g. the weights for generating different commands, and the stochastic distribution of command arguments, are constantly tinkered with. Such modifications are intermixed with observing their efficacy using the measurement mechanism. There are many attempts to mechanise command generation to meet different coverage criteria in literature, but we believe, it is also vitally important to provide the right programming mechanisms permitting a test model programmer to efficiently generate test suites with a good test coverage.

Iterative modelling of features.

The work on testing the aeternity blockchain began by modelling first the most core transactions, e.g., the spend transaction which transfers electronic money between account, and introducing the needed information (accounts) in the testing state. Then, iteratively, a group of transactions corresponding to a particular new feature (e.g. oracles) were introduced in the model. This required introducing new commands, and adding new information to the testing state. Moreover, pre-existing commands such as e.g. the command corresponding to executing a spend transaction were manually inspected to determine whether they needed

to be modified because of “feature interactions” with the new feature. This occurred, for example, because we wanted to generate, with some probability, spend transactions which send electronic money to accounts identified with oracles to test that functionality (a change to command generation). Moreover, we wanted to measure that the generation is successful, e.g., that generated spend transaction do attempt to transfer electronic money to oracle accounts, and that these generated transactions are valid. This constitutes a change in the measurement code, concretely tagging spend transactions with information regarding the provenance of the accounts in the transaction.

Relaxing postcondition checks.

For testing complex system there is often a choice between testing more exhaustively the result of each command (a stronger postcondition), or programming a less strong postcondition which will label less command results as being incorrect. For the aeternity blockchain, for instance, we can choose to carefully inspect all accounts after executing a transaction to determine that no electronic money was incorrectly transferred. For example, after a spend transaction which transfers funds between *A* and *B* we should verify that the transaction correctly modified the accounts of *A* and *B*, but, in principle, we should also inspect the blockchain to ensure that no other account *C* either received additional funds, or lost them. In this testing effort we instead use a very simple postcondition check, e.g., that transactions believed to be valid are indeed recognised as valid (and vice versa), not even bothering to check that e.g. the spend transactions correctly transfer money between *A* and *B*. We instead rely on the fact that later transactions which are believed to be valid (there are sufficient funds in the accounts) will be incorrectly rejected by the faulty blockchain as accounts contain too little funds. This strategy works because the model specifies the permitted money transfers, and updates the testing state accordingly. Moreover, this testing state is used to predict which transactions will be successful (valid) or not, depending on the the amount of money predicted in the accounts involved in the transaction.

There are obvious trade-offs here. By having weaker postcondition they take less time to check (we no longer have to inspect the entire blockchain after each transaction), and as a result, we can execute a larger number of tests. However, failing tests can become much longer, and harder to interpret, as the location where an error is detected can be quite far from the transaction which introduced the error. Moreover, we must be reasonable certain that accounts with too little/too much electronic money will eventually be identified during testing. Thus, great care has to be taken in practice to assign “reasonable” balances to initial accounts, and in general to use only a small number of accounts during testing.

Coping with partial knowledge.

A particular problem with testing the aeternity blockchain is that, paradoxically, to the model the execution of an aeternity transaction is not completely deterministic. This is because the exact amount of gas required to execute a transaction is hard to predict, and thus, the exact reduction of electronic money in the account of the user that pays for a transaction cannot be predicted exactly. This is a significant problem as we need to predict accurately which transactions are valid, e.g., that the paying user has enough electronic money to pay for the transaction. If we cannot determine exactly how much electronic money a user has, how can we accurately predict whether a user has sufficient funds to pay for a transaction?

Fortunately we can during testing separate the two different concerns, paying for gas usage, and transferring funds, by always transferring quantities of electronic money several orders of magnitude greater than the amount of money used for paying for gas usage. Our testing assumes that there is always enough money to pay for gas usage (i.e., we simply do not test that functionality of the block chain), whereas explicit electronic money transfers and payments are checked accurately (modulo the small indeterminate loss of funds due to gas usage).

Moreover, during testing we retrieve from the blockchain the exact amount of money which was spent to pay for gas usage during the postcondition checking of a transaction. Once a test case terminates, we check that the sum of money remaining in the blockchain (the sum of money in all accounts) is equal to the sum of money in all accounts at the start of the test case, minus the sum of electronic money which was spent to pay for gas for all valid transactions in the test case.

8 | RELATED WORK

As blockchains are instances of complex distributed systems, mainstream testing techniques to ensure the correct operations of such classes of systems are applicable to blockchains too. One prominent testing technique which has applied to test complex concurrent systems is fuzz testing^{27,31}.

In this paper, we have shown how EQC is used for functional testing. EQC generators are carefully designed in order to generate valid and semi-valid data. In a way, this is similar to black-box fuzz testing, a popular testing technique used for finding

security vulnerabilities in software applications. In fuzz testing, typically input data is modified, either randomly or through mutations, with the goal of generating semi-valid data. Such data is fed to the SUT in the hope that it crashes. A well-known fuzz testing tool is the American Fuzzy Lop (AFL)²⁸, which combines the use of genetic algorithms to mutate program input, with compile-time instrumenting to prioritise inputs that cause new program behaviour to appear. A number of hybrid fuzz testing tools have started to appear³², which combine “normal” fuzz testing with concolic testing³³, which use symbolic program execution and constraint solving techniques. The goal of these combined frameworks is to permit “deeper” program bugs to be found, i.e., where inputs need to be more finely structured (or sequential in nature) to exhibit program malfunction. The so called generation-based fuzzers³⁴, in contrast to fuzzers which are seeded with pre-existing program inputs, start from, like EQC specifications, a model of the system under test.

Although there are similarities between fuzz testing and EQC testing, there are a number of significant differences too. In EQC testing the same test model is used both to generate test data, and to evaluate test success, permitting very expressive program properties to be expressed succinctly and tested, whereas most fuzz testing frameworks focus on more obvious and pre-defined program failure conditions (crashes, buffer overflows, etc.). Recently, the Echidna tool³⁵ has been proposed for test generation of smart contracts written in Ethereum using fuzzing techniques. Echidna performs automatic test generation to detect violations of assertions and custom properties of smart contracts. Automatic test generation in Echidna is achieved using the Application Binary Interface (ABI) of Ethereum smart contracts. These tests allow the user to check assertion violations and user-defined predicates with minimal user effort. There are two main differences between the testing procedure followed in Echidna and ours: the test generation and the testing execution environment.

Our test generation procedure relies on the development of a model and the one used by Echidna uses the ABI of the contract, which is automatically provided by the compiler. The main disadvantage of using a model-based technique is that the development of the models can be time-consuming and requires extra effort from the user. Despite this drawback, providing a model unlocks the power of symbolic execution which gives useful information during test generation and can potentially make a difference in finding certain kinds of bugs. The model also allows further customisation in test generation like custom argument generators.

A number of papers focusing on testing blockchains or blockchain components are starting to appear. In³⁶, for instance, a platform is provided for IoT blockchain applications, with a focus on enabling regression testing. In³⁷, QuickCheck properties are presented for testing Patricia trees, a component of many blockchains. In³⁸ [an experiment in simulating and verifying a blockchain protocol, through an embedding in SDL](#).

In³⁹, property-based testing techniques are applied to the (abstract) Scorex modular blockchain framework⁴⁰. The properties tested focus on basic properties of most blockchains, e.g., testing that the blockchain only record successful operations, etc. In this article, apart from testing such basic blockchain soundness conditions, we test new transaction types supporting e.g. smart contracts, oracles, state channels, etc. The introduction of these new transactions risks causing unwanted “feature interactions”, and it is of vital importance to find such problems before the new transactions become widely used. Also, in this article, we provide a thorough description of the resulting blockchain testing framework, which, apart from EQC itself, is available as open source. In⁴¹ [the implementation of the smart contract programming language Scilla is checked using property-based testing techniques](#). For testing Tezos blockchain an OCaml property-based tool is used to test part of the blockchain functionality⁴². In comparison with these works on formally verifying and testing blockchains, some of them with the aid of property-based testing techniques, our present effort is rather more complete and targets the actual Aeternity implementation, not some abstracted blockchain protocol. The effort on testing the Scilla implementation is interesting, and a similar testing effort should be implemented for the Sophia runtime system (the principal smart contract programming language in Aeternity).

A blockchain related research area which is currently attracting considerable interest is smart contract analysis, [a survey of different formal verification approaches in the area can be found in](#)⁴³. Notable articles on e.g. estimating the cost of executing a contract (in gas) include⁴⁴, and, more generally, on formal verification of contract safety^{45,46} and security⁴⁷. In⁴⁸ [a smart contract certification framework based on the Coq proof assistant is presented, and](#)⁴⁹ [extends this work with support for property-based testing](#). In⁵⁰ John Hughes presents his work on testing Cardano smart contracts using Haskell QuickCheck.⁵¹ contains a general discussion on how testing blockchain based applications. Although delivering interesting results, these applications of property-based testing concern the testing and verification of particular smart contracts implemented on various blockchains, a topic quite different from the one of this article: how to test the transactional engine of the Aeternity blockchain.

The results provided in the literature study in³⁰, show an ongoing interest in researching the readability of test code in the software and testing community. The authors found that the majority of studies focused on “automatically generated unit test code in Java and some on manually created test code”. They also found that “the majority of studies investigate the influence of

individual factors on readability such as names, comments, summaries, or structural features". Thus, studying the readability of alternative testing techniques like EQC testing, is an interesting area of research when comparing different testing techniques. In this paper, we introduce the intuition that the way we structure the EQC model for testing improves the readability of the model, however, a formal study in line with the studies presented in³⁰ is needed.

9 | CONCLUSIONS AND FUTURE WORK

In this paper we provide an experience report on testing feature-rich blockchain code using property-based testing (PBT). In particular, we have used the PBT tool Quviq Erlang QuickCheck to test the blockchain transaction engine of the Aeternity blockchain platform which includes features such as a naming system, oracles, smart contracts and state channels. The strength of the approach is that, by collecting test statistics and tuning the test data distribution accordingly, automatically generating test cases provides better coverage than unit testing^{9,12,13}.

Of course, the main effort in PBT is creating the testing model. In our work the structure of this test model follows the structure of the blockchain, i.e., it cleanly separates different blockchain features (e.g., oracles, smart contracts) into different model parts, and moreover, reduces the amount of boilerplate test model code by focusing on the identification of valid blockchain transactions. The test model is evaluated through a careful instrumentation of test code which permits observations of which combinations of features have been tested during a test run, and with which frequency.

Property-based testing has been used in other quality sensitive domains, such as the automotive industry, telecommunication and health-care industry. The novelty described in this paper, apart from the application area itself, is in the use of production software as an oracle of software updates, and with regards to a number of test model techniques that make it possible to reduce failing test cases to minimal failing test cases.

The EQC specification is publicly available, and is used and improved together with changes to the Aeternity feature set. Most recently, two new transaction types have been added and the model has shown to be valuable in such a major refactoring. The ideas presented in that specification should carry over to any feature-rich blockchain.

We have deliberately chosen to access the software on a level that allows to test the core functionality without having to perform time consuming or non-deterministic behaviour. Had we chosen to test the implementation on a system level using, for example, several docker containers, then mining and transaction memory pools would have made the execution much slower and the results far less predictable.

Using this approach, testing a large number of scenarios is not difficult performance-wise. **On average, on a modest laptop, around 250 blockchain transactions was completed per second during testing using EQC.**

The test statistics, showing which blockchain features have been tested and how thoroughly testing has been, is a vital tool to judge the whether sufficient testing has been realised. Such test measures are partly added by the persons writing the EQC specification. However, this is only as powerful as the statistics one can come up with, and here there is clearly a lot of room for future improvements. **In future work, we plan to investigate how to automatically instrument test code with statistical information, and whether there are tools that calculate the efficacy of test runs.**

There are several improvements possible for the EQC specification itself. For example, to add support in the model to focus testing on a specific set of transactions. After changing the state channel part of the software, one may for instance want to run tests with many state channel transactions. The tweaking needed to do this is now a manual process, but a swarm testing⁵² approach could be beneficial here.

The test approach described in this article is applicable to other blockchains as well. **We have recently presented in¹⁵ some preliminary work on a method that uses property-based testing for testing the contract reentrancy vulnerability of smart contracts written in Solidity and running on the Ethereum blockchain platform.** The test models we use for the Ethereum blockchain strongly resemble the ones developed for Aeternity, which have been described in this article.

In summary, the property-based testing approach has been shown to be successful in finding faults in an efficient way. The more features a blockchain has, the more it can benefit from using property-based testing for additional quality assurance.

References

1. Nakamoto S. Bitcoin: A Peer-to-Peer Electronic Cash System. *Cryptography Mailing list at https://metzdowd.com* 2009.

2. Nakamoto S, others . Bitcoin. <https://bitcoin.org/en/>; 2019. Accessed: 2019-05-20.
3. Buterin V. Ethereum: A next-generation smart contract and decentralized application platform. <http://ethereum.org/ethereum.html>; 2013.
4. Goodman L. Tezos-a self-amending crypto-ledger. White paper. https://www.tezos.com/static/papers/white_paper.pdf; 2014.
5. Team TA. Aeternity. <https://aeternity.com/>; 2019. Accessed: 2019-05-20.
6. Ali M, Nelson J, Shea R, Freedman MJ. Blockstack: A Global Naming and Storage System Secured by Blockchains. In: USENIX Association; 2016; Denver, CO: 181–194.
7. Armstrong J. Erlang. *Commun. ACM* 2010; 53(9): 68–75. doi: 10.1145/1810891.1810910
8. Arts T, Hughes J, Johansson J, Wiger UT. Testing telecoms software with Quviq QuickCheck. In: Feeley M, Trinder PW, eds. *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006* ACM; 2006: 2–10
9. Arts T, Hughes J, Norell U, Svensson H. Testing AUTOSAR software with QuickCheck. In: IEEE Computer Society; 2015: 1–4
10. Claessen K, Hughes J. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In: ICFP '00. ACM; 2000; New York, NY, USA: 268–279
11. Claessen K, Hughes J. QuickCheck: a lightweight tool for random testing of Haskell programs. In: Odersky M, Wadler P, eds. *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*. ACM; 2000: 268–279
12. Hughes JM, Bolinder H. Testing a database for race conditions with QuickCheck: none. In: Rikitake K, Stenman E., eds. *Proceedings of the 10th ACM SIGPLAN workshop on Erlang, Tokyo, Japan, September 23, 2011* ACM; 2011: 72–77
13. Gay G, Staats M, Whalen MW, Heimdahl MPE. The Risks of Coverage-Directed Test Case Generation.. *IEEE Trans. Software Eng.* 2015.
14. Calder M, Kolberg M, Magill EH, Reiff-Marganiec S. Feature Interaction: A Critical Review and Considered Forecast. *Comput. Netw.* 2003; 41(1): 115–141. doi: 10.1016/S1389-1286(02)00352-3
15. Ballesteros I, Benac-Earle C, Bueso LE, Frelund LÅ, Herranz Á, Mariño J. Automatic generation of attacker contracts in Solidity. <https://easychair.org/smart-program/FLoc2022/paper2436.html>; 2022. Accessed: 2022-08-18.
16. Beizer B. *Software Testing Techniques (2nd Ed.)*. USA: Van Nostrand Reinhold Co. . 1990.
17. Merkle RC. A Digital Signature Based on a Conventional Encryption Function. In: Pomerance C., ed. *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*. 293 of *Lecture Notes in Computer Science*. Springer; 1987: 369–378
18. King S, Nadal S. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August 2012*; 19(1).
19. Service EN. <https://ens.domains/>; 2020. Accessed: 2020-03-04.
20. Keys R. <https://www.realitykeys.com/>; 2020. Accessed: 2020-03-04.
21. Wiger U. Building a Blockchain in Erlang | Code Mesh LDN 18. https://www.youtube.com/watch?v=I4_xX_Zs2eE&feature=youtu.be&t=1730; 2018.
22. Tromp J. Cuckoo Cycle: a Memory Bound Graph-Theoretic Proof-of-Work. In: Springer. ; 2015: 49–62.
23. Eyal I, Gencer AE, Sirer EG, Van Renesse R. Bitcoin-NG: A Scalable Blockchain Protocol. In: NSDI'16. USENIX Association; 2016; Berkeley, CA, USA: 45–59.

24. Syverson P. A taxonomy of replay attacks [cryptographic protocols]. In: IEEE. ; 1994: 187–191
25. Ivanov D. Aeternity State Channels. Talk at Blackseachain 2018. <https://www.youtube.com/watch?v=2zry6xOZI7g&feature=youtu.be>; 2018.
26. Arts T, Hughes J. How Well are Your Requirements Tested?. In: IEEE Computer Society; 2016: 244–254
27. Miller BP, Fredriksen L, So B. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 1990; 33(12): 32–44. doi: 10.1145/96267.96279
28. Zalewski M. American fuzzy lop. <http://lcamtuf.coredump.cx/afl/>; 2017. Accessed: 2019-05-22.
29. Mista A, Russo A, Hughes J. Branching Processes for QuickCheck Generators. *SIGPLAN Not.* 2018; 53(7): 1–13. doi: 10.1145/3299711.3242747
30. Winkler D, Urbanke P, Ramler R. What Do We Know About Readability of Test Code? - A Systematic Mapping Study. In: ; 2022: 1167-1174
31. Takanen A, Demott JD, Miller C. *Fuzzing for Software Security Testing and Quality Assurance*. Norwood, MA, USA: Artech House, Inc. 2nd ed. 2018.
32. Stephens N, Grosen J, Salls C, et al. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In: The Internet Society; 2016.
33. Godefroid P, Klarlund N, Sen K. DART: directed automated random testing. In: Sarkar V, Hall MW., eds. *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005* ACM; 2005: 213–223
34. Godefroid P, Kiezun A, Levin MY. Grammar-based whitebox fuzzing. In: Gupta R, Amarasinghe SP., eds. *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008* ACM; 2008: 206–215
35. Grieco G, Song W, Cygan A, Feist J, Groce A. Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts. In: ISSTA 2020. Association for Computing Machinery; 2020; New York, NY, USA: 557–560
36. Walker MA, Dubey A, Laszka A, Schmidt DC. PlaTIBART: A Platform for Transactive IoT Blockchain Applications with Repeatable Testing. In: M4IoT '17. ACM; 2017; New York, NY, USA: 17–22
37. Midtgaard J. QuickChecking Patricia Trees. In: Wang M, Owens S., eds. *Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers*. 10788 of *Lecture Notes in Computer Science*. Springer; 2017: 59–78
38. Duan Z, Mao H, Chen Z, Bai X, Hu K, Talpin JP. Formal Modeling and Verification of Blockchain System. In: ICCMS '18. Association for Computing Machinery; 2018; New York, NY, USA: 231-235
39. Chepurnoy A, Rathee M. Checking laws of the blockchain with property-based testing. In: IEEE; 2018: 40–47
40. Chepurnoy A, others . Scorex - The modular blockchain framework. <https://github.com/ScorexFoundation/Scorex>; 2019. Accessed: 2019-05-21.
41. Hoang T, Trunov A, Lampropoulos L, Sergey I. Random Testing of a Higher-Order Blockchain Language. In: ICFP '22. ACM; 2022; New York, NY, USA.
42. Overview of Testing in Tezos. <https://tezos.gitlab.io/developer/testing.html>; 2022. Accessed: 2022-08-18.
43. Tolmach P, Li Y, Lin SW, Liu Y, Li Z. A Survey of Smart Contract Formal Specification and Verification. *ACM Comput. Surv.* 2021; 54(7). doi: 10.1145/3464421
44. Albert E, Gordillo P, Rubio A, Sergey I. GASTAP: A Gas Analyzer for Smart Contracts. *CoRR* 2018; abs/1811.10403.

45. Bhargavan K, Delignat-Lavaud A, Fournet C, et al. Formal Verification of Smart Contracts: Short Paper. In: Murray TC, Stefan D., eds. *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016* ACM; 2016: 91–96
46. Amani S, Bégel M, Bortin M, Staples M. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In: CPP 2018. ACM; 2018; New York, NY, USA: 66–77
47. Lu N, Wang B, Zhang Y, Shi W, Esposito C. NeuCheck: A more practical Ethereum smart contract security analysis tool. *Software: Practice and Experience* 2021; 51(10): 2065-2084. doi: <https://doi.org/10.1002/spe.2745>
48. Annenkov D, Nielsen JB, Spitters B. ConCert: A Smart Contract Certification Framework in Coq. In: CPP 2020. Association for Computing Machinery; 2020; New York, NY, USA: 215–228
49. Annenkov D, Milo M, Nielsen JB, Spitters B. Verifying, testing and running smart contracts in ConCert. https://www.researchgate.net/publication/343472842_Verifying_testing_and_running_smart_contracts_in_ConCert; 2020.
50. Hughes J. Something good is always possible: testing smart contracts with QuickCheck. Talk at Craft Conference 2021. <https://www.youtube.com/watch?v=wtdBljSxEeI>; 2021.
51. Lal C, Marijan D. Blockchain Testing: Challenges, Techniques, and Research Directions. *CoRR* 2021; abs/2103.10074.
52. Groce A, Zhang C, Eide E, Chen Y, Regehr J. Swarm Testing. In: ISSTA 2012. ACM; 2012; New York, NY, USA: 78–88

