# EthVer: Formal verification of randomized Ethereum smart contracts, extended version

*Abstract*—

**Despite the great potential capabilities and the mature technological solutions, the smart contracts have never been used at a large scale, one of the reasons being the lack of good methods for verification of correctness and security of the contracts — although the technology itself (e.g. the Ethereum platform) is well studied and secure, the actual smart contracts are human-made and thus inherently error-prone. As a consequence, critical vulnerabilities in the contracts are discovered and exploited every few months. The most prominent example of a buggy contract was the infamous DAO attack — a successful attack on the largest Ethereum contract in June 2016 resulting in $70 mln-worth Ether stolen and the *hard fork* of the Ethereum network (80% of Ethereum users decided to revert the transaction and hence two parallel transaction histories exist from that event).**

**The main contribution of this work is the automatic method of formal verification of randomized Ethereum smart contracts. We formally define and implement the translation of the contracts into MDP (Markov decision process) formal models which can be verified using the PRISM model checker — a state of the art tool for formal verification of models. As a proof of concept, we use our tool, *EthVer*, to verify two smart contracts from the literature: the *Rock-Paper-Scissors* protocol from *K. Delmolino et al., Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab.* and the Micropay 1 protocol from *R. Pass, a. shelat, Micropayments for decentralized currencies*.**

*Index Terms*—**cryptocurrencies, ethereum, smart contracts, verification, formal methods, model checking**

CONTENTS

## I. Introduction

The notion of a *smart contract* was first introduced in 1997 by Nick Szabo [34]. The main idea behind smart contracts is that many contractual clauses (such as collateral, bonding, delineation of property rights, etc.) can be embedded in the hardware and software and smart contracts are protocols that can serve as digital agreements between the users of the network — the fulfillment of the agreement is automatically guaranteed by the design of the system instead of some external authority like banks, governments or courts.

The first practical implementations of smart contracts emerged together with the introduction of *Bitcoin* in 2009 [27], however they gained their popularity five years later when Ethereum [13] was announced — the first fully operational digital platform dedicated particularly for smart contracts, much more convenient to use and with much larger capabilities than Bitcoin.

Despite their huge potential capabilities, smart contracts have never been adopted on a large scale, one of the reasons being the fact that it is difficult to verify the correctness and security of the contract. As a consequence, critical vulnerabilities are discovered and exploited every few months [1], [3], [2]. The most prominent example of a buggy contract was the infamous DAO attack [18] — a successful attack on the largest Ethereum contract in June 2016 resulting in $70 mln-worth Ether stolen and the *hard fork* of the Ethereum network (80% of Ethereum users decided to revert the transaction and hence two parallel Ethereum blockchains exist from that event).

Several approaches to verification of smart contracts have been proposed, including the automatic and semi-automatic tools which analyze the contract code and checks if it satisfies some set of predefined security properties [26], [37], [28] or user-defined properties [24], [14], [23], [22], [8]. The other line of work focuses on providing tools to help creating smart contracts that follow some security patterns by design [29], [32], [36]. All these approaches suffers from the following limitation: they focus only on the security of the contract code without analyzing the outcome of the scenario of its usage. In other words, the traditional verification tools answer the questions: *Is this contract guaranteed to not "crash"? Can it end up in an unwanted state? Will the contract function be always executed till the end?* In contrast, none of the methods answers the question: *What will be the result if I use the contract in the following way?* Furthermore, we are not aware of any solution which verifies the randomized smart contracts and its probabilistic properties, for example: *Are the chances of winning in that lottery indeed equal to 1/2?*

### A. Our contribution

The main contribution of this work is an automatic method for verifying randomized protocols built on top of Ethereum smart contracts. We introduce the ETV language which allows to easily create such protocols using the syntax of Solidity (the main contract language of Ethereum). Furthermore, we formally define the translation of such protocols into the Markov decision processes (MDPs) which can be verified for security and correctness using the PRISM model checker — a state of the art tool for formal verification of models. The formal translation is accompanied by the implementation of *EthVer* — a fully operational compiler that translates an ETV program into a MDP in the PRISM syntax. As a proof of concept we use our tool to compile and verify two protocols from the literature: the Rock-Paper-Scissors protocol from *K. Delmolino et al., Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab.* [17] and the Micropay 1 protocol from *R. Pass, a. shelat, Micropayments for decentralized currencies* [30].

What distinguishes our work is that we model not only the smart contract code but also the *scenario* of its execution. The syntax of the contract part of ETV language is very similar to *Solidity* (the programming language of Ethereum smart contracts), while the syntax of the scenario part is very similar to *web3.js* JavaScript library (the library used to execute Ethereum smart contracts). This makes the usage of ETV intuitive to anyone familiar with Solidity and web3.js. We are not aware of any other tool that allows to verify the

scenario of smart contract execution in such a way. Also, to the best of our knowledge, our method is the first one that allows to verify the probabilistic properties of randomized smart contracts.

Another novel feature of the ETV language is the abstract construction for cryptographic commitments and digital signatures — the important cryptographic primitives present in many smart contracts. The *EthVer* compiler not only translates these abstract objects into MDP (which allows verification) but also provides the exact implementation of commitments and signatures in the syntax of Solidity and web3.js. This prevents the user from implementing them by hand, which might me cumbersome and error-prone. We are not aware of any other extension to Solidity which offers such functionality.

The full code of the *EthVer* compiler, as well as the extended version of this paper are available at the *EthVer* project page[1].

### B. Related work

Among the existing solutions for verification of smart contracts we can distinguish two main groups which can be summarized as the *verification approach* in which the contract is checked for compliance with some specification or security policy and the *design approach* which simplify the smart contract creation process by providing frameworks for their development. Below we analyze the related work falling into these two categories and describe how *EthVer* differs from the existing solutions.

*1) Verification approach:* This group contains static analysis tools for automated bug-finding [26], [37], [28] that verify the code for satisfying some pre-defined security properties, such as the correct order of transactions, timestamp dependency, prodigality or liveness. The other group of tools [23], [8], [22] provides semi-automatic methods for proving the contract-specific properties. These tools require some manual interaction from the user, such as specifying the loop invariants in the bytecode. Another work [12] analyzes Ethereum contracts by translating them into a functional language $F^*$. The language provides verification methods and an interactive proof assistant, however the translation supports only a part of the EVM syntax. Other solutions [21], [15] provide dynamic monitoring of the predefined security properties, such as transaction order dependency or *callback free executions*, a lack of which is claimed to be the source of common bugs. Both of these methods provide the defense from only a tiny subset of the possible vulnerabilities in the contracts.

All the solutions mentioned in the previous paragraph are able to analyze only the Ethereum Virtual Machine (EVM) pre-compiled bytecode. The other tool [24] analyzes the high-level Solidity contract code using symbolic model checking for the user-defined *policies*. However the policies are restricted to quantifier-free first-order logic, so it can only solve the state reachability problem and hence, e.g., cannot verify probabilistic properties. Another interesting approach [14] provides a game-theoretic framework in which the smart contract is translated into a concurrent game and the properties of this game are further analyzed using the novel method of abstraction-refinement. This method offers much lower computational complexity than the exact model checking of the whole model, however it does not provide the exact result of the verification, but only the lower and upper bound.

There are several other tools that provide static analysis for generic properties [35], [5], [4], [6], [7]. None of them is however accompanied by a scientific paper so the full specifications of the actual verification methods are hard to identify.

*2) Design approach:* One example of a high-level language that impose secure design of the smart contract is Simplicity [29]. It is however a general purpose language for smart contracts with no compiler to the EVM bytecode. Another interesting tool allows exporting the compiled code to the intermediate language WhyML[2] which in turn can be checked for security patterns using the program verification platform Why3 [19]. This tool however does not support the full range of properties to verify, in particular it cannot verify probabilistic properties. A slightly different approach [36] introduces *security patterns* — the best practices that must be met while developing the contract code, such as, e.g., performing calls at the end of a function. This approach however does not allow to specify custom properties to be satisfied by the contract.

### C. EthVer

The *EthVer* compiler falls somehow between the *verification approach* and *design approach* — it is able to verify the actual code of Ethereum smart contract (and also the scenario of its usage), however it requires the contract code to be written in the ETV language which is a slightly modified version of Solidity. Furthermore it allows to verify any custom property written in a dedicated language, including the probabilistic properties.

To the best of our knowledge, none of the solutions described in this section offers the exact model checking of the *probabilistic* properties of the randomized contract. Moreover, the existing approaches focus on verifying the *contract* without taking into account the pattern of execution of the contract by the users. Instead, in *EthVer* we verify the *protocol* which consists of the contract and the *scenario* of usage of the contract by the users. Hence, we are able to verify not only the correctness and security of the contract code, but also the *instructions* on how to use the contract.

It is worth noting here, that the two features of *EthVer* described above allows to perform the full formal verification of the *rock-paper-scissors* protocol [17] and the *Micropay 1* protocol [30], which cannot be done in any of the other tools analyzed in this section. We briefly describe this analysis in sec. VII.

## II. Preliminaries

### A. Ethereum languages

The actual code of Ethereum smart contracts is written in the machine code of Ethereum Virtual Machine (EVM).

[1] github.com/ethver/ethver

[2] https://why3.lri.fr

However, the platform provides several high level, user-friendly languages to write the code of a contract with the Solidity language being the most popular among them. The syntax of Solidity is based on JavaScript with some extra features added to handle the flow of money and cryptographic operations. Calling a contract function is realized by sending a special transaction to the contract address. There are several convenient GUI tools to deploy and execute smart contracts, such as, e.g., a desktop application Ethereum Wallet or a web application Remix as well as the console client `geth`[3]. Under the hood they all use the JavaScript API with the web3.js library[4] which provides the basic functions to interact with the contract as well as some cryptographic functions widely used in smart contracts (such as hash functions and digital signatures). The main web3.js function to interact with a contract is the `sendTransaction` method which is called on a contract function object and takes as arguments the arguments to the function and the sender address (the `from:` field) as well as the *value* attached to the transaction. The example usage of the `sendTransaction` function is listed below:

```
Bank.deposit.sendTransaction(1,
{from: "0x14723a09acff6d2a60dcdf7aa4aff308fddc160c",
  value: web3.utils.toWei("5", "finney")});
```

Note that the transaction value must be passed as an integer number of *wei* (1 *wei* $= 10^{-18}$ *ether*), however, the `web3.utils.toWei` function can be used to easily convert from different units like *finney* (1 *finney* = 0.001 *ether*).

### B. The PRISM model checker

PRISM is a probabilistic model checker, a tool originally described in [25]. It is designed for formal modeling and analysis of systems which present random or probabilistic behavior. Many smart contracts fit into this category, so we decided to use PRISM as the backbone for our formal verification of Ethereum smart contracts.

PRISM supports different types of models, including discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs), Markov decision processes (MDPs), probabilistic automata (PAs), probabilistic timed automata (PTAs). In *EthVer* we decided to use Markov decision processes, since they allow non-determinism and hence are the best fit for randomized protocols built on top of smart contracts.

The PRISM model is defined as a set of states and transitions between them. Each transition is represented with a *command* of form

$$[label]\ \ guards\ \texttt{->}\ updates\texttt{;}$$

where *guards* are the conditions needed to be met in order for the transition to be *enabled*, *updates* represent the probabilistic choices in the algorithm, and *label* is an optional identifier of the transition used for synchronization. The syntax of the updates is as follows:

$$p_1\ \texttt{:}\ update_1\ +\ p_2\ \texttt{:}\ update_2\ +\ \ldots\ +\ p_n\ \texttt{:}\ update_n$$

The updates list reflects the situation when several transitions are possible from the same state and the choice of the actual transition is probabilistic: the $i$-th transition happens with probability $p_i$ and results in $update_i$ of the model variables.

For more detailed introduction to PRISM please refer to appendix A of the extended version of the paper[5].

### C. Cryptographic commitments

A cryptographic commitment scheme is a protocol which consists of two phases: *commit* and *open* (the second phase is also referred to as the *reveal* phase). In the most common implementation during the *commit* phase the user chooses a value $r$ to which they will be committed, chooses a random value $s$ and computes $c = H(r, s)$ where $H$ is a hash function (a collision-resilient function that is hard to invert). Then the user publishes the hash $c$ while keeping $r$ and $s$ secret.

In the *open* phase the user reveals the chosen values $r$ and $s$ and anyone can use $c$ to verify if the author of the commitment didn't change $r$. The cryptographic commitments are *hiding* and *binding* in the sense that:

- the value of $c$ reveals no information about $r$,
- once the values of $r$ and $c$ are fixed, it is infeasible to come up with another value of $r$ which matches the same $c$.

Ethereum provides a convenient way to implement the cryptographic commitments using *SHA-3* function (also known as *keccak256*):

```
hash = web3.utils.sha3(web3.utils.
 toHex(r).substr(2)+web3.utils.toHex(s))
```

Listing 1. Computing the commitment in web3.js v1.0.0

```
uint8 r; string s; bytes32 c =
 keccak256(abi.encodePacked(48+r,s))
```

Listing 2. Computing the commitment in Solidity v0.5.2

Note the different names of the hash function and the subtle differences in passing the arguments. These differences follow from different APIs used by Solidity and web3.js, but the underlying hash function is the same.

### D. Digital signatures

Ethereum supports digital signatures based on elliptic curve cryptography implementing the SECP-256k1 standard as described in [16]. The web3.js library provides two useful functions: `web3.eth.accounts.sign(m, sk)` for signing and `web3.eth.accounts.recover(m, signature)` for recovering the public key of the author of the signature. Solidity provides the function `ecrecover` (hash, v, r, s) which takes the hash of the messages and $(v, r, s)$ values, which

[3]ethereum.org

[4]web3js.readthedocs.io

[5]Recall that the extended version of the paper as well as the code of *EthVer* and example contracts are available at github.com/ethver/ethver

are the 3 parts of the `signature`[6]. In the most common scenario the signatures are created off-chain in web3.js and then they are later verified by the contract. Due to different APIs of Solidity and web3.js, a special care is needed for the format of numbers passed to the `sign` and `recover` functions. Listings 3 and 4 show an example code for signing a message in web3.js and verifying the obtained signature in Solidity[7].

```
r2_ = web3.utils.toHex(r2).substr(2);
concat = c + web3.utils.padLeft(r2_, 2)
  + a.toLowerCase().substr(2);
msg = "msg" + web3.utils.sha3(concat);
s = web3.eth.accounts.sign(msg, privKey);
```

Listing 3. Signing a message in web3.js v1.0.0

```
string header = "\x19Ethereum
 Signed Message:\n69msg0x";
bytes data = hexToBytes(keccak256(
 abi.encodePacked(c,r2,a)));
bytes32 msgHash = keccak256(
 abi.encodePacked(header, data));
return ecrecover(msgHash,
 s_v, s_r, s_s) == a))
```

Listing 4. Verifying the signature in Solidity v0.5.2

## III. Interacting with the contract

The code of a smart contract does not carry all the information needed for verification. Consider a simple Bank contract written in Solidity:

```
contract Bank {
  uint balance;

  function deposit() public payable {
    balance = balance + msg.value;}

  function withdraw(uint amount) public {
    if (amount <= balance) {
      balance = balance - amount;
      msg.sender.transfer(amount);}}}
```

Listing 5. A simple Bank contract

Is this a secure smart contract? The answer to this question depends on how we want to use the contract and what behavior of the contract is expected. For example, this contract can be considered secure if we want a bank in which anyone can deposit money and then anyone can withdraw it. On the other hand, if we define the security of a bank with the rule that *only the person who has deposited the money can withdraw it*, then of course this contract is not secure.

In order to concretize the requirements for the contract we must formulate the *scenario* and the *properties* which we want to be satisfied. In case of the Bank contract they can be as follows:

*Scenario:*

- User $A$ deposits 10 *finney*[8].
- User $A$ withdraws 10 *finney*.

*Properties:*

- User $A$ gets back his deposited 10 *finney*.

Of course this property is not always satisfied, which can be shown using the *counterexample* scenario in which user $A$ deposits 10 *finney* and then user $B$ withdraws the same 10 *finney*. After that the user $A$ no longer can withdraw 10 *finney*, since the contract account is already empty.

Although this scenario of using the Bank contract may look artificial (why not to use the contract in a different way?), in case of many contracts, the scenario of the proper usage is obvious and well defined. Consider, e.g., a simple lottery in which user $A$ bets 10 *finney* and wins 20 *finney* with probability $1/2$ (otherwise loses). In such case, the scenario and the properties are as follows:

*Scenario:*

- User $A$ deposits 10 *finney*.
- User $A$ waits for the result of the lottery.

*Properties:*

- With probability 1/2: the user $A$ receives the reward of 20 *finney*.
- With probability 1/2: the user $A$ receives nothing.

## IV. The ETV language

To model in a verifiable way the contract together with the scenario of its usage we introduce the ETV language. The ETV program consists of two parts: the first part is a slightly modified code of the actual Solidity smart contract, while the second one represents the scenario using web3.js commands.

### A. Bounded integers

The main issue with the verification of smart contracts in PRISM is the usage of *bounded integers* in PRISM. The reason for it is that a new state in the PRISM model is created for every valuation of the variables, and thus increasing the range of variables increases the number of states in the model in the exponential way. On the other hand, in Solidity/web3.js the smallest type for storing integers is `uint8` which is capable of storing numbers from the range $[0, 255]$. Frequently we use such type to store the variables which can have only a small number of different values (e.g., only 0, 1 or 2) and we do not need the whole range of `uint8`.

Because of this limitation we introduce in ETV the *bounded integer* type `uint(N)` which in practice is the main difference between the syntax of ETV and Solidity/web3.js.

### B. Communication

A protocol can contain some operations that are performed directly between the parties of the protocol (without interaction with the blockchain), for example exchanging hashes. We define a dedicated *communication* section for such operations in

[6]Note that we don't describe here how to sign messages in Solidity. In fact, Solidity does not provide convenient API for this. The reason is that a private key is required to sign and we rarely want to do this in the contract code, because we do not want to reveal the private keys to public.

[7]This is the actual code of computing and verifying the signature $\sigma = \mathsf{sig}(c, r_2, a)$ from the Micropay 1 protocol (cf. sec. VII).

[8]Recall that 1 *finney* = 0.001 ETH is a denomination of Ether, the currency of Ethereum. For simplicity, we neglect the transaction fees, unless stated otherwise.

the ETV language. Such approach allows us to properly model the *adversarial* behavior by allowing the malicious party to execute the commands from the *communication* section.

### C. Cryptographic primitives

The other important feature of the ETV language is the abstract syntax for cryptographic primitives, such as hashes, commitments and signatures. Such primitives can be (a) translated into PRISM which allows to verify the properties of the contract and (b) translated into the actual implementation in Solidity and web3.js which reduces the probability that the user implements it incorrectly. The last feature is especially important because the current versions of Ethereum programming languages (Solidity v0.5.2 and web3.js v1.0.0) present large differences in the API for the cryptographic functions and a special care must be taken to make sure that the Solidity part and the web3.js part of the code operates on the same numbers[9].

## V. The compiler

The main practical result of this work is the implementation of *EthVer* — a compiler written in Haskell that takes as the input an ETV file (let us call it `example.etv`) and produces:

- `example.sol` — the contract code in Solidity which can be directly deployed to the Ethereum blockchain,
- `example.scen` — the scenario of the execution of the contract containing the exact JavaScript web3.js commands which can be directly used to execute the contract,
- `example.prism` — the PRISM Markov decision process (MDP), which can be directly used in the PRISM model checker.

While translating ETV code to Solidity and web3.js is straightforward, the translation from ETV to PRISM MDP is highly nontrivial and was the main challange during creation of *EthVer*. In the next section we describe the core concepts behind this translation and their implementation in *EthVer*.

Furthermore, in the extended version of the paper we formally define the full syntax and the semantics of the ETV language (appendices B and C). In appendix D we formally define the translation from ETV to MDP and prove that it preserves the semantics of ETV.

## VI. Modeling the protocol as Markov decision process

In this section we present the core concepts of *EthVer* — the way in which we translate an ETV program into a Markov decision process (MDP).

### A. Modeling the contract execution

We model the honest execution of the contract using 4 PRISM modules[10]: `player0`, `player1`, `contract`, `blockchain`. We show the role of each module on the example of the simple Bank contract (cf. listing 5) and the following scenario of its usage:

- User A deposits 10 *finney*.
- User A withdraws 10 *finney*.

We model the honest execution of the scenario with the following commands in the `player0` module:

```
module player0
[broadcast_deposit] (state0 = 1) -> (state0' = 2)
  & (deposit_value0' = 10);
[broadcast_withdraw] (state0 = 2) -> (state0' = 3)
  & (withdraw_amount0' = 10);
endmodule
```

Each command sets the value as well as all the arguments of the function call and then triggers the corresponding commands in `contract` and `blockchain` modules using the PRISM synchronization mechanism — the command with a non-empty label (the string in square brackets) can be executed only in parallel with the corresponding function with the same label in other modules (as long as such command exists in other modules).

The actual process of calling the contract function consists of two phases: in the first phase, in parallel to `[broadcast_*]` command from `player0` module, PRISM executes the synchronized command from the `blockchain` module which stores the information that this function call is now in the *broadcast* state. Then at some later point PRISM can take one of the function calls from the *broadcast* state and actually execute the corresponding contract code. This is accomplished by another pair of synchronized commands from `blockchain` and `contract` modules[11].

### B. Modeling the adversary

Although the verification of the honest execution of the protocol is important, we frequently face vulnerabilities in the contracts which reveal themselves only when one (or more) of the participants misbehave, i.e., deviate from following the scenario. In order to model the adversarial player, we decided to give them the ability to interact with the contract in an arbitrary way. More concretely, the adversary can **call any function of the contract, in any order, with any arguments, as many times as wanted**. With such definition of the adversary we can model any 2-player contract[12] in one of the 3 following *modes*:

- *honest* mode — honest player 0 vs honest player 1
- *adversarial player 0* mode — malicious player 0 vs honest player 1
- *adversarial player 1* mode — honest player 0 vs malicious player 1

[9]Examples of the syntax of commitments and signatures in Solidity and web3.js have already been presented in sec. II-C and II-D

[10]A PRISM model can consists of several modules, each corresponding to a different part of the system and each with a separate set of variables.

[11]The same pattern of a two-phase function execution could be accomplished using only the `player0` and `contract` modules, however because of visibility of the variables in PRISM, the `blockchain` module is needed to correctly pass the arguments of the call.

[12]The current version of *EthVer* is limited to 2-players protocols only. However, all the security claims as well as the formal translation defined in appendix D of the extended version of the paper hold also for protocols with larger number of players. Note that although *EthVer* accepts only 2-player protocols, it verifies the contract also against the attacks in which more adversarial players join the protocol at the same time.

### C. Modeling the communication

As it was already discussed in sec. IV-B, some protocols contain phases in which the players do off-chain computation and exchange the computed numbers without calling the contract. Since these procedures do not involve the actual execution of the contract code, they should not be handled in the same way as the contract calls are. On the other hand, we do not want to limit the capabilities of an adversarial player, and hence we need to give the adversary the possibility of performing these procedures at any time, with any arguments (like in case of contract calls). We model every such action as a *communication function* that are called during honest scenario execution and can also be freely called by the adversary.

These communication functions are stored in the separate `communication` section of the ETV code. The syntax of such functions is very similar to the syntax of contract functions, with the only difference that it cannot handle the money transfers. These functions translate to the `communication` module in the PRISM code which can be triggered using the PRISM synchronization mechanism from the player modules in a similar fashion to the contract function (but without going through the *broadcast* state and without involving the `blockchain` module).

### D. Modeling the cryptographic commitments

Recall that in the standard implementation of random commitments (sec. II-C) during the *commit* phase two random numbers ($r$ and $s$) are generated and then they are later revealed during the *open* phase. Since all variables in PRISM are *public*, we cannot just store $r$ and $s$ as PRISM variables, because it will break the *hiding* property of the commitment. It follows from the fact that MDPs are non-deterministic and for MDPs we always compute the maximal (or minimal) probability `Pmax/Pmin`, where the probability is computed over all the random choices, while the maximum (minimum) is taken over all the non-deterministic choices of the model and hence the non-deterministic choices must be done before the random choices..

To best illustrate the problem, consider a simple game in which $A$ creates a commitment by choosing $r$ and $s$ at random and then $B$ tries to guess $r$ before the revealing phase. If we store the value of $r$ in a variable right after the *commit* phase, then the automaton that models $B$ can non-deterministically choose the correct value of $r$ (since now there is no more randomness in the protocol) and win the game. Hence, in order to properly model the real behavior of keeping $r$ secret, we need to not store the final value of $r$ during the *commit* phase and postpone the actual random choice until the *open* phase.

In our implementation each commitment in PRISM can be in one of the following states: *init*, *committed*, or *revealed*. All the commitments start in the *init* state. During the honest scenario, when the player *creates a random commitment*, the appropriate variable switches to the *committed* state, but no actual choice of the value is made. During the *open* phase, the player needs to call a separate `revealCmt` method which performs the actual random choice. After this call, the commitment variable switches from the *committed* state to the *revealed* state.

Using the same mechanism the adversary can either commit to a random value (by switching to the *committed* state and postponing the actual choice until the revealing phase) or he can immediately commit to the value of his choice by switching directly to the *revealed* state. In both cases he cannot later change the chosen value. This models the real implementation of the commitments in which the chosen value also cannot be changed after the commitment is created.

This approach is realized in *EthVer* by providing the `cmt_uint` type and functions `randomCmt` and `verCmt` which implement directly the described functionality.

### E. Modeling the digital signatures

We introduce a templated type for the signatures: `signature(T1, T2, ...)`. The types `T1`, `T2`, ... are the types of the *fields* of the signature — the values that we want to sign (when we want to sign more than one value, we usually concatenate them before signing). For the signature type we provide two constructions to create and to verify the signature:

```
sigma = sign(c, r2, a);
verSig(verAddress, sigma, (c, r2, a));
```

We model the signatures in PRISM in the following way:

- each signature is initially in the *init* state,
- there is a separate PRISM variable for each *field* of the signature as well as for the address of the author of the signature,
- whenever a signature is created, the fields are assigned with the values being signed (and the author's address); these fields' values cannot be later changed.

The adversary is not allowed to change any field of the existing signature. However to not limit their ability to *interact with the contract in any way at any time*, we allow them to freely create new signatures, i.e., to sign any data at any time with their own key.

### F. Modeling the time

Solidity natively supports creating contracts dependent on time using the `now` variable. Moreover, we introduce in ETV the `wait(condition, time)` statement which implements the conditional wait: after reaching that point of the scenario, the party pauses the execution of the protocol until the condition is satisfied or a particular time has passed.

We model the time in PRISM using the `time_elapsed` variable and the synchronized commands labeled `[time_step]`. The `time_elapsed` counter is increased in either of the two cases:

- all the honest parties have finished all of their allowed scenarios steps and are waiting on the `wait` statement,
- the honest party has finished all of their allowed scenarios steps while the adversary decides to not execute any step.

This reflects the assumption that the honest parties always follow the protocol and execute every scenario step within a

given time limit while the adversary can interrupt the protocol at any time and refuse to execute a scenario step within the time limit.

## VII. Case study: verification of two protocols from the literature

As a proof of concept we use the *EthVer* compiler to formally verify two protocols from the literature: The Rock-Paper-Scissors protocol from *K. Delmolino et al., Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab.* [17], and The Micropay 1 protocol from *R. Pass, a. shelat, Micropayments for decentralized currencies* [30]. The results of verification of these protocols are broadly described in appendices E and F of the extended version of this paper.

In the first work [17] the authors analyze the actual smart contract for the Rock-Paper-Scissors game which was created by undergrad students during the cryptocurrency lab. The authors point out several typical mistakes that were made during designing this contract and present a few good programming practices to avoid such bugs in the future. Using *EthVer* we were able to automatically find all the contract bugs and fix all of them. The case study is iterative — after each bug fix we rerun the verification and every time the *EthVer* shows us the next bug of the protocol. Using this iterative method we implement the total of 6 bug fixes which finally lead us to the correct version of the contract.

It must be stressed, that in the original paper the authors analyze the contract and fix all the bugs *by hand*. In contrast, in our experiment the only manual action is that we rewrite the original contract to the ETV language (which requires only a few minor tweaks) and then *EthVer automatically* finds all the bugs and provides the *witness* for each of them which make it easy to fix the bugs.

In the second paper [30] the authors present the Micropay 1 protocol — a smart contract which can serve as a platform for *micropayments* — fast and cheap off-chain transactions which only occasionally require the interaction with the blockchain. In the original version of the paper [30] (published on the *22nd ACM CCS '15 conference*) the authors describe a buggy version of the protocol — the contract is vulnerable to so called *front-running attack*. After the publication we discovered the bug (it was also discovered independently by Joseph Bonneau) and contacted the authors with our findings. As a result, they published the corrected version of the paper [31]. In case of this protocol we also were able to find and fix the bug using *EthVer*. Again, the *EthVer* has found the bug *automatically*, which means that if the authors verified the protocol before publication using *EthVer* or a similar tool, they would discover the attack and the buggy version of the contract would never be published.

Our case study involved 9 tested models in total (7 versions of the RPS contract and 2 versions of the Micropay contract[13]).

[13]The ETV code of all tested models is available in the project repository, github.com/ethver/ethver.

The table I shows the performance of all the test runs. Each test was performed on a laptop with Intel Core i7-4750HQ CPU @ 2.00 GHz and 8 GB RAM.

| protocol | number of states | model checking time |
|---|---|---|
| rps v1 | 1.6M | 130s |
| rps v1a | 1.2M | 72s |
| rps v1b | 1.9M | 75s |
| rps v2 | 0.8M | 66s |
| rps v3 | 6.6M | 470s |
| rps v3a | 5.3M | 264s |
| rps v4 | 5.2M | 238s |
| micropay v1 | 16M | 24min |
| micropay v2 | 490M | 124min |

TABLE I
Performance of all the test runs

## VIII. Conclusions

In this work we present the *EthVer* compiler — a novel tool for formal verification of Ethereum smart contracts. We have developed a dedicated ETV language for designing secure and verifiable contracts. We have formally defined and proved the correctness of the translation of this language to Markov decision process (MDP) models in PRISM. This translation has been implemented in Haskell and works as a standalone computer program.

The novelty of our approach lies in the three features: (1) the verification of the whole cryptographic protocol consisting of a smart contract and a scenario of its execution, (2) the verification of the probabilistic properties of the randomized contracts, and (3) the abstract language construction for cryptographic commitments and signatures, which can be automatically translated into the actual Ethereum code and into the PRISM model. To the best of our knowledge, no other verification approach offers any of these 3 functionalities.

The automatic verification of the model generated by *EthVer* is possible due to our original method of modeling the contract as MDP. Our technique allows to verify the correctness and security of the honest execution of the protocol and also verifies the protocol against the attacks of the adversarial user. Moreover, in case the vulnerability of the protocol is found, our tool returns the *witness* — the execution path which leads to the undesired state of the protocol.

As a proof of concept we used *EthVer* to verify two smart contracts from the literature. In both cases *EthVer* was able to automatically find the bugs that were claimed to be found manually by the authors. This means that the vulnerable contracts would not have been created if the authors had used *EthVer* for their verification.

The experiments results show that the verification is practical — it can be performed on a medium-class PC within a reasonable time frame. However, the experiments revealed also the inherent limitation of our method — the size of the model (and hence the verification time) grows exponentially with the number of parameters of the contract. Therefore our method is most suitable for the contracts of a limited size — for larger models the exact model checking is not possible and other verification methods must be used.

## References


[1] Accidental bug may have frozen $280 million worth of digital coin ether in a cryptocurrency wallet, accessed: 2019-03-02, https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html

[2] How to find $10m just by reading the blockchain, accessed: 2019-03-02, https://coinspectator.com/news/539/how-to-find-10m-just-by-reading-the-blockchain

[3] An in-depth look at the parity multisig bug, accessed: 2019-03-02, http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/

[4] Manticore, https://github.com/trailofbits/manticore

[5] Mythril, https://github.com/ConsenSys/mythril

[6] Smartcheck, https://github.com/smartdec/smartcheck

[7] solgraph, https://github.com/raineorshine/solgraph

[8] Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying ethereum smart contract bytecode in isabelle/hol. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 66–77. CPP 2018, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3167084, http://doi.acm.org/10.1145/3167084

[9] Baier, C.: On algorithmic verification methods for probabilistic systems. Ph.D. thesis, habilitation thesis, University of Mannheim (1998)

[10] Baier, C., Kwiatkowska, M.: Model checking for a probabilistic branching time logic with fairness. Distributed Computing **11**(3), 125–155 (1998)

[11] Bartoletti, M., Galletta, L., Murgia, M.: A minimal core calculus for solidity contracts. In: Data Privacy Management, Cryptocurrencies and Blockchain Technology, pp. 233–243. Springer (2019)

[12] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., et al.: Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security. pp. 91–96. ACM (2016)

[13] Buterin, V.: Ethereum: A next-generation smart contract and decentralized application platform (2014), https://github.com/ethereum/wiki/wiki/White-Paper, accessed: 2016-08-22

[14] Chatterjee, K., Goharshady, A.K., Velner, Y.: Quantitative analysis of smart contracts. In: European Symposium on Programming. pp. 739–767. Springer, Cham (2018)

[15] Cook, T., Latham, A., Lee, J.H.: Dappguard: Active monitoring and defense for solidity smart contracts. Retrieved July **18**, 2018 (2017)

[16] Courtois, N.T., Grajek, M., Naik, R.: Optimizing sha256 in bitcoin mining. In: Kotulski, Z., Księżopolski, B., Mazur, K. (eds.) Cryptography and Security Systems. pp. 131–144. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)

[17] Delmolino, K., Arnett, M., Kosba, A.E., Miller, A., Shi, E.: Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D.S., Brenner, M., Rohloff, K. (eds.) FC 2016 Workshops. Lecture Notes in Computer Science, vol. 9604, pp. 79–94. Springer, Heidelberg, Germany, Christ Church, Barbados (Feb 26, 2016). https://doi.org/10.1007/978-3-662-53357-4_6

[18] Falkon, S.: The story of the dao — its history and consequences (2017), https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee

[19] Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems. pp. 125–128. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

[20] Goldreich, O.: Foundations of Cryptography: Basic Applications, vol. 2. Cambridge University Press, Cambridge, UK (2004)

[21] Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzky, N., Sagiv, M., Zohar, Y.: Online detection of effectively callback free objects with applications to smart contracts. Proceedings of the ACM on Programming Languages **2**(POPL), 1–28 (2017)

[22] Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., Roşu, G.: Kevm: A complete formal semantics of the ethereum virtual machine. pp. 204–217 (07 2018). https://doi.org/10.1109/CSF.2018.00022

[23] Hirai, Y.: Defining the ethereum virtual machine for interactive theorem provers. In: Financial Cryptography Workshops (2017)

[24] Kalra, S., Goel, S., Dhawan, M., Sharma, S.: Zeus: Analyzing safety of smart contracts. In: NDSS. pp. 1–12 (2018)

[25] Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proc. 23rd International Conference on Computer Aided Verification (CAV'11). LNCS, vol. 6806, pp. 585–591. Springer (2011)

[26] Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269. ACM (2016)

[27] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system, http://bitcoin.org/bitcoin.pdf

[28] Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference. pp. 653–663 (2018)

[29] O'Connor, R.: Simplicity: A new language for blockchains. In: Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security. pp. 107–120 (2017)

[30] Pass, R., shelat, a.: Micropayments for decentralized currencies. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS 2015: 22nd Conference on Computer and Communications Security. pp. 207–218. ACM Press, Denver, CO, USA (Oct 12–16, 2015). https://doi.org/10.1145/2810103.2813713

[31] Pass, R., shelat, a.: Micropayments for decentralized currencies. Cryptology ePrint Archive, Report 2016/332 (2016), http://eprint.iacr.org/2016/332

[32] Pettersson, J., Edström, R.: Safer smart contracts through type-driven development. Master's thesis. Chalmers University of Technology, Sweden (2016)

[33] Puterman, M.L.: Markov decision processes: discrete stochastic dynamic programming. John Wiley & Sons (2014)

[34] Szabo, N.: Formalizing and securing relationships on public networks. First Monday **2**(9) (1997)

[35] Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.: Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 67–82. CCS '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3243734.3243780, http://doi.acm.org/10.1145/3243734.3243780

[36] Wohrer, M., Zdun, U.: Smart contracts: Security patterns in the ethereum ecosystem and solidity. In: 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE). pp. 2–8. IEEE (2018)

[37] Zhou, E., Hua, S., Pi, B., Sun, J., Nomura, Y., Yamashita, K., Kurihara, H.: Security assurance for smart contract. In: 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS). pp. 1–5. IEEE (2018)


## Appendix

### A. PRISM model checker

The main functionality and syntax of PRISM were already described in the main part of the paper. In this appendix we present the more advanced features of PRISM.

*1) Synchronization:* PRISM provides a useful method for synchronization. Every PRISM model can be split into several *modules*. Commands from different modules can be synchronized if they share the same *label*. Consider the following example:

```
module m1
s : [0..3] init 0;
d : [0..4] init 0;

[choice] s=0 -> (s'=1);
[choice] s=0 -> (s'=2);
[] s=1 -> 0.5 : (s'=3)&(d'=1) + 0.5 : (s'=3)&(d'=2);
[] s=2 -> 0.5 : (s'=3)&(d'=3) + 0.5 : (s'=3)&(d'=4);
endmodule

module m2
t : [0..1] init 0;

[choice] t=0 -> (t'=1);
endmodule
```

Each *labeled* command of a module can be executed only simultaneously with a matching command with the same label from every **other** module (if it contains any). In the example above, the first command of module `m1`, i.e., `[choice] s=0 -> (s'=1)` can be executed only simultaneously with the command `[choice] t=0 -> (t'=1)` from the module `m2`. Analogously the second command of `m1`, i.e., `[choice] s=0 -> (s'=2)` also can be executed only simultaneously with the command `[choice] t=0 -> (t'=1)` from the module `m2`. Therefore it is, e.g., not possible that `m1` moves to the state with $(s \neq 0)$ while `m2` remains in the $(t = 0)$ state.

*2) Properties:* Once the model is defined, PRISM can (a) simulate the model, (b) verify some properties of the model. While simulating the model is a convenient tool to analyze the model by hand, the second feature (verifying the properties) is crucial for this work and serves as a main building block for the verification of smart contracts. The properties need to be specified in a special *PRISM property specification language*. This language subsumes several well-known probabilistic temporal logics, including PCTL, CSL, probabilistic LTL and PCTL*. PRISM also supports most of the (non-probabilistic) temporal logic CTL. For a detailed description of the *PRISM property specification language*, please refer to the official documentation on the project page[14].

One of the properties which we can formulate for our dice example is:

```
P=? [ F s=7 & d=3 ]
```

which can be read as: *What is the probability of reaching from the initial state the state where* $(s = 7)$ *and* $(d = 3)$*?* In other words: *What is the probability that the protocol finishes successfully and the result is 3?* We can enter this property and verify the model as a DTMC (discrete-time Markov chain) in PRISM — the result turns out to be equal to 1/6.

*3) Nondeterminism:* Smart contracts usually are nondeterministic — the user is free to choose which function of the contract to execute and what arguments to pass to the function. Because of that we decided to model the contract as a Markov decision processes (MDP) [33] which is a model type that allows nondeterminism.

**Definition 1** (Markov decision process)**.** *A Markov decision process (MDP) is a tuple* $\mathcal{M} = (S, s_0, Act, P)$*, where* $S$ *is a finite set of states,* $s_0$ *is the initial state, Act is a finite set of* actions *and* $P : S \times Act \times S \to [0, 1]$ *is a function such that* $\sum_{s' \in S} P(s, \alpha, s') \in \{0, 1\}$ *for each state-action pair* $(s, \alpha)$*. We denote the set* $\{\alpha | \sum_{s' \in S} P(s, \alpha, s') = 1\}$ *by the set of actions* enabled *in state* $s$*.*

Note that in a particular state, multiple actions can be enabled at the same time and hence MDPs allow nondeterministic transitions.

In PRISM the MDP actions are represented by PRISM *commands*. Consider the following example model:

[14]www.prismmodelchecker.org

```
s : [0..3] init 0;
d : [0..4] init 0;

[] s=0 -> (s'=1);
[] s=0 -> (s'=2);
[] s=1 -> 0.5 : (s'=3)&(d'=1) + 0.5 : (s'=3)&(d'=2);
[] s=2 -> 0.5 : (s'=3)&(d'=3) + 0.5 : (s'=3)&(d'=4);
```

In this model from the initial state $(s = 0)$ two nondeterministic transitions are possible: to state $(s = 1)$ or $(s = 2)$. Depending on the chosen transition, the final probability distribution for the result $d$ is different: in the first case $d$ is chosen randomly between 1 and 2 and in the second case it is chosen between 3 and 4. For the nondeterministic models we cannot verify the `P=?` properties (like for the deterministic ones), since the probability depends on the nondeterministic choices. Instead, for the nondeterministic models we can verify the properties of form:

```
Pmin=? [ F s=3 & d=3 ]
Pmax=? [ F s=3 & d=3 ]
```

Using such properties we can compute the *minimum probability* and the *maximum probability* of reaching some state in the sense that **the minimum/maximum is taken over all the nondeterministic choices of the model** while **the probability is computed over all the probabilistic choices of the model**. If we verify the given properties for the model described above, the results will be 0 and $1/2$ respectively. This means that the minimum probability of reaching $(d = 3)$ is 0 — the pessimistic scenario is the execution of the first command (`s=0 -> (s'=1)`), in which case we cannot reach $(d = 3)$, regardless of *luck* in the second transition. The maximum probability of reaching $(d = 3)$ is 0.5, because in the optimistic scenario we choose the second transition (`s=0 -> (s'=2)`) and then we have $50\%$ chances of reaching $(d = 3)$ in the second step.

*4) Witness/counterexample:* PRISM provides a very useful feature of generating witness/counterexample for *reachability properties*. If the probability verification described in the previous paragraph gives us the answer that does not match the desired behavior of the model, we usually want to know what is this pessimistic (or optimistic) scenario which minimizes (maximizes) the `Pmin` (or `Pmax`) value. Using *reachability properties* we can usually define the unwanted state of the model and find the path which leads to this state. Using the same example model, we can formulate the property:

```
E [ F (s = 3) & (d != 3) & (d != 4) ]
```

which can be read as: *Is it possible to reach the state in which* $(s = 3)$ *and* $d$ *is different from 3 and 4?* In case of the analyzed model this property turns out to be true and PRISM generates a *witness* for this property — the path with all transitions chosen prior to reaching the given state.

The feature of generating witness/counterexample is a priceless tool for debugging smart contracts. When a vulnerability in a contract exists, usually it means that there exists some adversarial behavior which leads to the unwanted state. Using

the *witness* feature of PRISM we can find the path leading to this state and fix the bug.

*5) Fairness:* Sometimes the model checking requires *fairness* to be taken into account [10], [9]. Fairness property means that *every command that can be executed infinitely many times, will eventually be executed*. Consider the following model:

```
s : [0..1] init 0;
t : [0..1] init 0;

[] s=0 -> true;
[] t=0 -> (t'=1);
```

This model can execute the first command in the infinite loop and never execute the second one. Fairness in this case means that since the second command is always enabled, it will be eventually executed. PRISM offers the verification of nondeterministic models with and without fairness enabled. In case of this example model we can verify if the model *will always reach the state* $(t = 1)$ using the property:

```
Pmin=? [ F t = 1 ]
```

The result of verification of this property in PRISM depends on whether we enable fairness or not: without fairness, the result is 0, while with fairness it is 1.

The fairness is crucial when modeling smart contracts. When two users concurrently call for a contract execution, the contract *decides* nondeterministically, which call will be executed first. We can imagine a theoretical scenario in which an honest user waits for the execution of his contract call, while a malicious user spams the contract with infinite loop of his calls to the contract. In practice, the calls to the contract are queued, so fairness assumption in this case is reasonable — once we call a contract, we can assume that the call will be eventually executed.

It is worth to note that non-probabilistic CTL model checking (like verification of *witness/counterexample* reachability properties) is not supported with fairness. Hence, in practice we will always use PRISM in one of two modes when verifying a smart contract:

- with fairness enabled in case of the verification of the actual contract properties (`Pmin`/`Pmax` PRISM properties),
- with fairness disabled in case of reachability properties (`E [...]` PRISM properties).

### B. The lexical structure of the ETV language

We formulate the syntax of the ETV language using BNF (Bachus-Naur form) notation. Then the *BNF Converter*[15] tool was used to create the lexer, the parser and the documentation for the language. The *EthVer* compiler is written in Haskell and uses the lexer and the parser generated by the *BNF Converter*. Below we present the abstract syntax of the ETV language produced by the *BNF Converter*.

*1) Identifiers:* Identifiers ⟨*Ident*⟩ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters `_ '`, reserved words excluded.

[15]bnfc.digitalgrammars.com

*2) Reserved words and symbols:* The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in ethver are the following:

```
address     bool                cash
cmt_uint    communication       constructor
contract    else                false
finney      function            hash
hashOf      if                  mapping
nonce       payable             public
random      randomCmt           revealCmt
scenario    sendCommunication   sendTransaction
sign        signature           this
transfer    true                uint
user        value               valueOf
verCmt      verSig              wait
while
```

The symbols used in ethver are the following:

```
;     =           {
}     [           ]
(     =>          )
,     .           ||
&&    ==          !=
<     <=          >
>=    +           -
*     /           %
!     msg.value   msg.sender
:
```

*3) Comments:* Single-line comments begin with `//`. Multiple-line comments are enclosed with `/*` and `*/`.

*4) The syntactic structure of ethver:* Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and $\epsilon$ (empty rule) belong to the BNF notation. All other symbols are terminals.

⟨*Program*⟩ ::= ⟨*ListUserDecl*⟩ ⟨*ListConstantDecl*⟩ ⟨*Contract*⟩ ⟨*Communication*⟩ ⟨*ListScenario*⟩

⟨*ListUserDecl*⟩ ::= $\epsilon$
| ⟨*UserDecl*⟩ `;` ⟨*ListUserDecl*⟩

⟨*ListConstantDecl*⟩ ::= $\epsilon$
| ⟨*ConstantDecl*⟩ `;` ⟨*ListConstantDecl*⟩

⟨*ListScenario*⟩ ::= $\epsilon$
| ⟨*Scenario*⟩ ⟨*ListScenario*⟩

⟨*UserDecl*⟩ ::= `user` ⟨*String*⟩

⟨*ConstantDecl*⟩ ::= ⟨*Ident*⟩ `=` ⟨*Integer*⟩

⟨*Contract*⟩ ::= `contract` ⟨*Ident*⟩ `{` ⟨*ListDecl*⟩ ⟨*Constructor*⟩ ⟨*ListFunction*⟩ `}`

⟨*Decl*⟩ ::= ⟨*Type*⟩ ⟨*Ident*⟩
| ⟨*Type*⟩ ⟨*Ident*⟩ `=` ⟨*Exp*⟩
| ⟨*Type*⟩ ⟨*Ident*⟩ `[` ⟨*Integer*⟩ `]`
| `mapping (address =>` ⟨*Type*⟩ `)` ⟨*Ident*⟩

⟨*ListDecl*⟩ ::= $\epsilon$
| ⟨*Decl*⟩ `;` ⟨*ListDecl*⟩

⟨*ListFunction*⟩ ::= $\epsilon$
| ⟨*Function*⟩ ⟨*ListFunction*⟩

⟨*Constructor*⟩ ::= `constructor ( ) {` ⟨*ListStm*⟩ `}`

⟨*Communication*⟩ ::= `communication {` ⟨*ListDecl*⟩ ⟨*ListFunction*⟩ `}`

⟨*Scenario*⟩ ::= `scenario` ⟨*Ident*⟩ `{` ⟨*ListDecl*⟩ ⟨*ListStm*⟩ `}`

⟨*Function*⟩ ::= `function` ⟨*Ident*⟩ `(` ⟨*ListArg*⟩ `)` `public {` ⟨*ListStm*⟩ `}`
| `function` ⟨*Ident*⟩ `(` ⟨*ListArg*⟩ `)` `public payable {` ⟨*ListStm*⟩ `}`

⟨*Arg*⟩ ::= ⟨*Type*⟩ ⟨*Ident*⟩

⟨*ListArg*⟩ ::= $\epsilon$
| ⟨*Arg*⟩
| ⟨*Arg*⟩ `,` ⟨*ListArg*⟩

⟨*Stm*⟩ ::= `{` ⟨*ListStm*⟩ `}`
| ⟨*Ident*⟩ = ⟨*Exp*⟩ `;`
| ⟨*Ident*⟩ `[` ⟨*Exp*⟩ `]` = ⟨*Exp*⟩ `;`
| `if (` ⟨*Exp*⟩ `)` ⟨*Stm*⟩
| `if (` ⟨*Exp*⟩ `)` ⟨*Stm*⟩ `else` ⟨*Stm*⟩
| `while (` ⟨*Exp*⟩ `)` ⟨*Stm*⟩
| ⟨*Exp*⟩ `. transfer (` ⟨*Exp*⟩ `) ;`
| ⟨*Exp*⟩ `. sendTransaction (` ⟨*ListCallArg*⟩ `) ;`
| ⟨*Exp*⟩ `. sendCommunication (` ⟨*ListExp*⟩ `) ;`
| ⟨*Exp*⟩ `. randomCmt ( ) ;`
| ⟨*Exp*⟩ `. revealCmt ( ) ;`
| `wait (` ⟨*Exp*⟩ `,` ⟨*Exp*⟩ `) ;`

⟨*ListStm*⟩ ::= $\epsilon$
| ⟨*Stm*⟩ ⟨*ListStm*⟩

⟨*ListExp*⟩ ::= $\epsilon$
| ⟨*Exp*⟩
| ⟨*Exp*⟩ `,` ⟨*ListExp*⟩

⟨*Exp*⟩ ::= ⟨*Exp*⟩ || ⟨*Exp1*⟩
| ⟨*Exp1*⟩

⟨*Exp1*⟩ ::= ⟨*Exp1*⟩ `&&` ⟨*Exp2*⟩
| ⟨*Exp2*⟩

⟨*Exp2*⟩ ::= ⟨*Exp2*⟩ == ⟨*Exp3*⟩
| ⟨*Exp2*⟩ `!=` ⟨*Exp3*⟩
| ⟨*Exp3*⟩

⟨*Exp3*⟩ ::= ⟨*Exp3*⟩ < ⟨*Exp4*⟩
| ⟨*Exp3*⟩ <= ⟨*Exp4*⟩
| ⟨*Exp3*⟩ > ⟨*Exp4*⟩
| ⟨*Exp3*⟩ >= ⟨*Exp4*⟩
| ⟨*Exp4*⟩

⟨*Exp4*⟩ ::= ⟨*Exp4*⟩ + ⟨*Exp5*⟩
| ⟨*Exp4*⟩ − ⟨*Exp5*⟩
| ⟨*Exp5*⟩

⟨*Exp5*⟩ ::= ⟨*Exp5*⟩ * ⟨*Exp6*⟩
| ⟨*Exp5*⟩ / ⟨*Exp6*⟩
| ⟨*Exp5*⟩ % ⟨*Exp6*⟩
| ⟨*Exp6*⟩

⟨*Exp6*⟩ ::= − ⟨*Exp6*⟩
| `!` ⟨*Exp6*⟩
| `random (` ⟨*Exp6*⟩ `)`
| `sign (` ⟨*ListExp*⟩ `)`
| `verSig (` ⟨*Exp*⟩ `,` ⟨*Exp*⟩ `, (` ⟨*ListExp*⟩ `) )`
| `verCmt (` ⟨*Exp*⟩ `,` ⟨*Exp*⟩ `)`
| `finney (` ⟨*Integer*⟩ `)`
| ⟨*Exp7*⟩

⟨*Exp7*⟩ ::= ⟨*Ident*⟩ `[` ⟨*Exp*⟩ `]`
| `valueOf (` ⟨*Exp7*⟩ `)`
| `hashOf (` ⟨*Exp7*⟩ `)`
| ⟨*Exp8*⟩

⟨*Exp8*⟩ ::= `msg.value`
| `msg.sender`
| ⟨*Ident*⟩
| ⟨*String*⟩
| ⟨*Integer*⟩
| `true`
| `false`
| `this`
| `(` ⟨*Exp*⟩ `)`

⟨*ListCallArg*⟩ ::= $\epsilon$
| ⟨*CallArg*⟩
| ⟨*CallArg*⟩ `,` ⟨*ListCallArg*⟩

⟨*CallArg*⟩ ::= ⟨*Exp*⟩
| `{ value :` ⟨*Exp*⟩ `}`

⟨*Type*⟩ ::= `uint (` ⟨*Integer*⟩ `)`
| `cash (` ⟨*Integer*⟩ `)`
| `bool`
| `cmt_uint (` ⟨*Integer*⟩ `)`
| `signature (` ⟨*ListType*⟩ `)`
| `address`
| `nonce`
| `hash`

⟨*ListType*⟩ ::= $\epsilon$
| ⟨*Type*⟩
| ⟨*Type*⟩ `,` ⟨*ListType*⟩

### C. The operational semantics of the ETV language

In this section we present the operational semantics of the ETV language. We follow the notation from [11] and define the semantics in a big-step style. The semantics of the ETV program is given by transitions from state to state. A state $\rho$ consists of six finite maps:

$$\rho = (F_{\mathsf{comm}}, F_{\mathsf{contr}}, V, C, S, U)$$

containing respectively: the communication functions definitions, the contract functions definitions, the current values of the variables, commitments and signatures and the user balances.

We use the standard notation $\{v_1/k_1, \ldots, v_n/k_n\}$ to represent a finite map which maps keys $k_i$ to values $v_i$ for $i \in \{1, 2, \ldots, n\}$. Moreover, we will denote by $\rho.V(k)$ the value associated to key $k$ in map $V$ and we will use the $\rho.V\{v_i/k_i\}$ notation to represent the map $V$ updated with the $(k_i, v_i)$ key-value pair.

We also define (following [11]) the auxiliary operators $\rho + \mathcal{X} : u$ and $\rho - \mathcal{X} : u$ which increase/decrease the balance of $\mathcal{X}$ by $u$, where $\mathcal{X}$ can be the contract $\mathcal{C}$ or a user.

All variables in ETV are strongly typed. At the moment of declaration, the type-default value is assigned to the variable. To improve readability, the declarations and the type checking are omitted in the descriptions of the semantics.

Initially, the state $\rho$ consists of:

- the map $F_{\mathsf{comm}}$ containing the definitions of all the communication functions,
- the map $F_{\mathsf{contr}}$ containing the definitions of all the contract functions,

- the map $V$ containing all the variables declared in the code. It also contains two special variables with keys `sender` and `value` to store the caller of the communication/contract function and the value of the contract call, respectively,
- the map $C$ containing all the commitment variables occurring in the code,
- the map $S$ containing all the signature variables occurring in the code,
- the map $U$ containing the initial balances of the users as well as the special value $U(\mathcal{C})$ indicating the initial contract balance.

We start defining the semantics of the ETV program by defining the semantics of a scenario and then recursively defining it for each subcomponents. We define the operational semantics of a scenario as follows:

$$\frac{}{[[\mathsf{scenario}(\mathit{name}, \mathit{decls}, \mathit{stms})]]_\rho = [[\mathit{stms}]]_\rho}$$

In the next sections we present the detailed semantics of all the statements and expressions. Note that we use the same notation to define the semantics of statements and expressions, however, for an expression $E$ the symbol $[[E]]_\rho$ returns a value (expressions have no side-effects) while for a statement $S$ the symbol $[[S]]_\rho$ returns the state after the execution of $S$. It should be always clear from the context whether the notation concerns a statement or an expression.

*1) Semantics of standard statements and expressions:* The semantics of most statements and expressions is identical to the other popular imperative languages like C or Java. Below we present the semantics of these statements and expressions in the big-step style notation.

*a) Statements:*

$$\frac{I = \langle \mathit{Ident} \rangle \quad [[E]]_\rho = v}{[[I = E]]_\rho = \rho.V\{I/k\}}$$

$$\frac{I = \langle \mathit{Ident} \rangle \quad [[E_1]]_\rho = i \quad [[E_2]]_\rho = v}{[[I[E_1] = E_2]]_\rho = \rho.V\{v/(I||i)\}}$$

$$\frac{[[E]]_\rho = b \in \{\mathtt{true}, \mathtt{false}\}}{[[\mathtt{if}\ (E)\ S_{\mathtt{true}}\ \mathtt{else}\ S_{\mathtt{false}}]]_\rho = [[S_b]]_\rho}$$

$$\frac{[[E]]_\rho = \mathtt{true}}{[[\mathtt{if}\ (E)\ S]]_\rho = [[S]]_\rho} \qquad \frac{[[E]]_\rho = \mathtt{false}}{[[\mathtt{if}\ (E)\ S]]_\rho = \rho}$$

$$\frac{[[E]]_\rho = \mathtt{false}}{[[\mathtt{while}\ (E)\ S]]_\rho = \rho}$$

$$\frac{[[E]]_\rho = \mathtt{true} \quad [[S]]_\rho = \rho'}{[[\mathtt{while}\ (E)\ S]]_\rho = [[\mathtt{while}\ (E)\ S]]_{\rho'}}$$

*b) Expressions:*

$$\frac{E \in \{\langle \mathit{Integer} \rangle, \langle \mathit{String} \rangle, \mathtt{true}, \mathtt{false}\}}{[[E]]_\rho = E} \qquad \frac{E = \mathtt{finney}(x)}{[[E]]_\rho = x}$$

$$\frac{I = \langle \mathit{Ident} \rangle}{[[I]]_\rho = \rho.V(I)} \qquad \frac{I = \langle \mathit{Ident} \rangle \quad [[E]]_\rho = v}{[[I[E]]]_\rho = \rho.V(I||v)}$$

$$\frac{}{[[\mathtt{this}]]_\rho = \rho.V(\mathtt{sender})}$$

$$\frac{}{[[\mathtt{msg.sender}]]_\rho = \rho.V(\mathtt{sender})}$$

$$\frac{}{[[\mathtt{msg.value}]]_\rho = \rho.V(\mathtt{value})}$$

$$\frac{}{[[(E)]]_\rho = [[E]]_\rho} \qquad \frac{\mathsf{op} \in \{-, !\}}{[[\mathsf{op}E]]_\rho = \mathsf{op}[[E]]_\rho}$$

$$\frac{\mathsf{op} \in \{+, -, *, /, \%, ||, \&\&, ==, != , <, <=, >, >=\}}{[[E_1 \mathsf{op}_2 E_2]]_\rho = [[E_1]]_\rho \mathsf{op} [[E_2]]_\rho}$$

*2) Semantics of the probabilistic statements:* We will use the following notation for the probabilistic assignment in which the variable $a$ is assigned the value $b_1$ with probability $p_1$, the value $b_2$ with probability $p_2$, etc.:

$$a = \begin{cases} p_1 : b_1 \\ \vdots \\ p_n : b_n \end{cases}$$

Using this notation we can define the semantics of the random expression:

$$[[\mathtt{random}(R)]]_\rho = \begin{cases} 1/R : 0 \\ \vdots \\ 1/R : (R-1) \end{cases}$$

*3) Semantics of the function execution:* The semantics of execution of the communication/contract function and the transfer of money is defined by the semantics of `transfer`, `sendCommunication` and `sendTransaction` methods presented below. The $\mathcal{A}.\mathtt{transfer}(u)$ statement transfers $u$ money from the contract to the user $\mathcal{A}$ as long as there are sufficient funds in the contract:

$$\frac{u \leq \rho.U(\mathcal{C})}{[[\mathcal{A}.\mathtt{transfer}(u)]]_\rho = \rho - \mathcal{C} : u + \mathcal{A} : u}$$

$$\frac{u > \rho.U(\mathcal{C})}{[[\mathcal{A}.\mathtt{transfer}(u)]]_\rho = \rho}$$

The $f.\mathtt{sendCommunication}(E_1, \ldots, E_n)$ statement calls the communication function $f$ with the values of the expressions $E_1, \ldots, E_n$ passed as arguments:

$$\frac{\begin{array}{c}[[E_1, \ldots, E_n]]_\rho = (v_1, \ldots, v_n) \\ f(\mathit{arg}_1, \ldots, \mathit{arg}_n)\{S\} \in F_{\mathsf{comm}} \\ \rho' = \rho.V\{v_1/\mathit{arg}_1, \ldots, v_n/\mathit{arg}_n\}\end{array}}{[[f.\mathtt{sendCommunication}(E_1, \ldots, E_n)]]_\rho = [[S]]_{\rho'}}$$

The $f.\mathtt{sendTransaction}(E_1, \ldots, E_n, \{\mathtt{value} : E\})$ statement checks if the caller has sufficient funds and calls the $f$ contract function with the values of the expressions

$E_1, \ldots, E_n$ passed as arguments and with $E$ money units attached as the call value:

$$\frac{\begin{array}{l}[[E_1, \ldots, E_n]]_\rho = (v_1, \ldots, v_n) \quad \rho.V(\texttt{sender}) = \mathcal{A} \\ [[E]]_\rho = u \leq \rho.U(\mathcal{A}) \quad f(arg_1, \ldots, arg_n)\{S\} \in F_{\mathsf{contr}} \\ \rho' = \rho - \mathcal{A} : u + \mathcal{C} : u \\ \rho'' = \rho'.V\{u/\texttt{value}, v_1/arg_1, \ldots, v_n/arg_n\}\end{array}}{\begin{array}{l}[[f.\texttt{sendTransaction}(E_1, \ldots, E_n, \{\texttt{value} : E\})]]_\rho \\ = [[S]]_{\rho''}\end{array}}$$

$$\frac{\rho.V(\texttt{sender}) = \mathcal{A} \quad [[E]]_\rho > \rho.U(\mathcal{A})}{[[f.\texttt{sendTransaction}(E_1, \ldots, E_n, \{\texttt{value} : E\})]]_\rho = \rho}$$

*4) Semantics of the `wait` statement:* The $\texttt{wait}(E_1, E_2)$ statement blocks the program execution until (a) the $E_1$ condition evaluates to true or (b) the $E_2$ units of time has elapsed. After either of the conditions is satisfied, the program execution is unblocked and the next statement is executed:

$$\frac{[[E_1]]_\rho = \texttt{true}}{[[\texttt{wait}(E_1, E_2); S]]_\rho = [[S]]_\rho}$$

$$\frac{\texttt{<time>} \geq [[E_2]]_\rho}{[[\texttt{wait}(E_1, E_2); S]]_\rho = [[S]]_\rho}$$

The `<time>` symbol above indicates the time (in seconds) that has elapsed since the beginning of the contract.

*5) Semantics of the cryptographic commitments:* We store the information about the commitments in the $\rho.C$ map. Each element of the map consists of the three fields: state, value and hash, where state $\in \{\mathsf{init}, \mathsf{committed}, \mathsf{revealed}\}$.

The ETV syntax provides 5 symbols related to cryptographic commitments: `randomCmt`, `revealCmt`, `valueOf`, `hashOf` and `verCmt`. Below we describe the semantics of each of these symbols.

The user can create a random commitment using the `randomCmt` function. We define the semantics of the `randomCmt` function as follows:

$$\frac{\rho.C(c).\mathsf{state} = \mathsf{init}}{[[c.\texttt{randomCmt}\,\texttt{()}]]_\rho = \begin{cases} 1/R : \rho.C\{0/c.\mathsf{value}, \mathsf{committed}/c.\mathsf{state}, \\ \qquad \mathsf{genHash}(c)/c.\mathsf{hash}\} \\ \vdots \\ 1/R : \rho.C\{(R-1)/c.\mathsf{value}, \mathsf{committed}/c.\mathsf{state}, \\ \qquad \mathsf{genHash}(c)/c.\mathsf{hash}\} \end{cases}}$$

When a commitment is created, its value is chosen at random and stored in $\rho.C$. The state of the commitment is set to committed. Moreover we define the abstract algorithm genHash, which generates the hash of the commitment and satisfies 2 conditions:

- the probability that it returns the same hash twice (even for the same commitment) is negligible,
- it is irreversible, i.e., given genHash(c) it is infeasible to learn $c$.

A commitment can be opened with the `revealCmt` function which changes the commitment state from committed to revealed:

$$\frac{\rho.C(c).\mathsf{state} = \mathsf{committed}}{[[c.\texttt{revealCmt}\,\texttt{()}]]_\rho = \rho.C\{\mathsf{revealed}/c.\mathsf{state}\}}$$

As long as the commitment is in the revealed state, the value of the commitment can be retrieved with the `valueOf` function :

$$\frac{\rho.C(c).\mathsf{state} = \mathsf{revealed}}{[[\texttt{valueOf}(c)]] = \rho.C(c).\mathsf{value}}$$

Similarly, the hash of the commitment can be retrieved using `hashOf` function in case the commitment is in the committed or revealed state:

$$\frac{\rho.C(c).\mathsf{state} \in \{\mathsf{committed}, \mathsf{revealed}\}}{[[\texttt{hashOf}(c)]] = \rho.C(c).\mathsf{hash}}$$

Finally, the correctness of a commitment can be verified using `verCmt` function by verifying if its hash is equal to the given hash $h$:

$$\frac{\rho.C(c).\mathsf{state} \in \{\mathsf{committed}, \mathsf{revealed}\}}{[[\texttt{verCmt}(c, h)]] = (\rho.C(c).\mathsf{hash} = h)}$$

In addition to the semantics described above, we require that the commitments satisfy the *hiding* and *binding* properties[16]:

- *hiding* — no one can learn the value of the commitment before the *reveal* phase,
- *binding* — after the *commit* phase the value of the commitment cannot be changed.

*6) Semantics of the digital signatures:* We store the information about all the digital signatures in the program in the $\rho.S$ map. For every $s$, $\rho.S(s)$ consists of the author of the signature and the values that are signed[17]. We define the semantics of the `sign` function as follows:

$$\frac{}{\begin{array}{l}[[s = \texttt{sign}(x_1, \ldots, x_n)]]_\rho \\ = \rho.S\{(\rho.V(\texttt{sender}), x_1, \ldots, x_n)/s\}\end{array}}$$

Since the signed data and the author of the signature are stored in the $\rho.S$ map, in order to verify if the signature is correct it suffices to check if the values stored in the map are the same as the values against which we verify the signature. Hence, we define the semantics of the `verSig` function as follows:

$$\frac{\rho.S(s) = (\mathcal{A}, x_1, \ldots, x_n)}{[[\texttt{verSig}(\mathcal{A}, s, x_1, \ldots, x_n)]]_\rho = \texttt{true}}$$

$$\frac{\rho.S(s) \neq (\mathcal{A}, x_1, \ldots, x_n)}{[[\texttt{verSig}(\mathcal{A}, s, x_1, \ldots, x_n)]]_\rho = \texttt{false}}$$

Following [20] we require from digital signature scheme to satisfy the three properties:

1) Each user can efficiently produce its own signature on data of its choice.

[16] cf. sec. II-C

[17] The number of the values being signed is inferred from the type of the $s$ variable

2) Every user can efficiently verify whether a given string is a signature of another (specific) user on a specific data.
3) It is infeasible to produce signatures of other users to data they did not sign.

The first two properties are clearly satisfied by our signature scheme. In addition, we require from the signatures to be *unforgeable*, i.e., we require from their implementation to satisfy the last property.

### D. Translation from ETV to MDP

The main part of the EthVer code is responsible for the translation from ETV language to Markov Decision Process (MDP) PRISM models. The full translation is defined in Haskell and is over 3000 lines of code long[18]. In this chapter we present the mathematical definitions of the most important parts of the translation.

According to the syntax described in sec. B, the ETV program consists of 5 elements:

$$program = (users, constants, contract, communication, scenarios)$$

Such program is translated by *EthVer* into a PRISM model consisting of the following modules:

- `blockchain`
- `contract`
- `communication`
- `player0`
- `player1`
- ⋮

In the following sections we define how each of these modules is created during the translation. We also prove that the translation we define preserves the semantics of the ETV language.

The current implementation of EthVer supports only two-party protocols, i.e., the ETV programs with two scenarios, but it can be easily extended to support protocols with larger number of parties. However, the translation described in this section as well as all the proven properties of it work for any number of players.

It is also worth noting that although we are limited only to two-party protocols, due to the adversary mode (cf. sec. VI-B) we can also model the attacks in which more adversarial players join the protocol.

*1) Notation:* The actual translation is defined as a Haskell program which takes as an input the ETV code and generates the PRISM commands for each piece of the code. In this chapter we present a mathematical definitions of the most important parts of this translation. We also prove that this transaction preserves the semantics of the ETV language. To describe the translation we recursively define the $\mathsf{T}()$ function which *compiles* the elements of the ETV language into PRISM commands. For each symbol from ETV we define the $\mathsf{T}()$ function using a special pseudocode which can contain:

- `mod`.**add**(...) function to add a PRISM command to module `mod`,
- **return** statement (only in case of compiling the expressions),
- statements that modify the state of the compiler,
- special variables `mod`, `sender`, `value`, `balance`, `balance0`, `balance1`, ... denoting respectively the currente module, the sender of the current call of the statement/function, the value of the current transaction, the contract balance and the balances of the users.

For each new element compiled, the compiler creates some new states in the PRISM module and adds PRISM *commands* which are the transitions between these states. For each *module*, the states are numbered separately with the consecutive integer numbers and the compiler provides access to the two special functions: $\sigma_{\mathsf{curr}}()$ which returns the final state of the last created command, and $\sigma_{\mathsf{new}}()$ which returns the first *unused* number of state which can be used to create a new state. After adding the command, $\sigma_{\mathsf{curr}}()$ is changed to the final state of the added command and $\sigma_{\mathsf{new}}()$ is changed to the next unused state number. Initially, $\sigma_{\mathsf{curr}}() = 1$ and $\sigma_{\mathsf{new}}() = 2$.

Furthermore, we use the following notation:

`mod`.**add**(
  **label** : *cmdLabel*,
  **from** : *fromState*,
  **to** : *toState*,
  **guards** : $[g_1, \ldots, g_n]$,
  **updates** : $[(i_1, e_1), \ldots, (i_1, e_m)]$)

for the function which adds to the `mod` a typical PRISM command[19]:

```
[cmdLabel]
  (stateVar = fromState)
& (g1)
...
& (gn)
->
  (stateVar' = toState)
& (i1' = e1)
...
& (ik' = ek);
```

where *stateVar* is the variable used for the states of a given PRISM module (`contrstate`, `commstate`, `state0`, `state1`, ... respectively). Each argument of the `mod`.**add**

[18]The part of the compiler code responsible for compilation from ETV to MDP is contained in the files `CompilerPrismEthver.hs`, `CodePrismEthver.hs`, `AuxPrismEthver.hs`, `ExpPrismEthver.hs`, `AuxWorldPrismEthver.hs`, `WorldPrismEthver.hs` in the project repository, github.com/ethver/ethver.

[19]Recall that the *guards* are the conditions that must be met to activate the command, while *updates* are the assignments that are performed after the execution of the command

function is optional. Moreover, we use the following syntax for the updates in probabilistic commands[20]:

$$\begin{aligned}&\mathsf{updates}:\{\\&\quad 1/R:[(i_{11},e_{11}),\ldots,(i_{1m_1},e_{1m_1})],\\&\quad\vdots\\&\quad 1/R:[(i_{R1},e_{R1}),\ldots,(i_{Rm_R},e_{Rm_R})]\}\end{aligned}$$

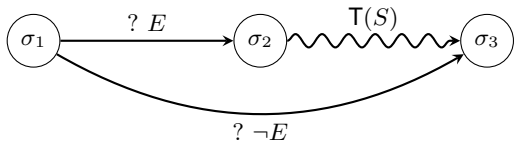
Thus, an example command which simulates the coin toss can be written using our syntax as follows:

$$\begin{aligned}&\texttt{mod}.\mathsf{add}(\\&\quad \mathsf{from}:\sigma_{\mathsf{curr}}(),\mathsf{to}:\sigma_{\mathsf{new}}(),\\&\quad \mathsf{updates}:\{1/2:[(r,0)],1/2:[(r,1)]\})\end{aligned}$$

This pseudocode corresponds to the following PRISM command:

```
[]
  (stateVar = currState)
->
  0.5 : (stateVar' = newState) & (r' = 0)
+ 0.5 : (stateVar' = newState) & (r' = 1);
```

In addition to the pseudocode of the translation we will frequently present the generated PRISM models as graphs:




In such graph the nodes correspond to the states of the PRISM model and the straight arrows correspond to the transitions between the states. Each transition can be decorated with a label (used for synchronization), guards (starting with '?'), or updates (starting with '!'). Furthermore, we will use the wavy arrows to indicate a larger subgraphs without the exact definition in the figure (e.g., the subgraphs which are the results of the translation of a different statement).

In the following sections we define the $\mathsf{T}$ function, starting from $\mathsf{T}(\textit{program})$ and then recursively defining it for all subcomponents. Note that we use the same notation for translating statements and expressions although the semantics is different: $\mathsf{T}(stm)$ modifies the state of the compiler and returns no value, while $\mathsf{T}(exp)$ returns the value of the expression and has now side effect. This should not be confusing since it always follows from context, which of the two meanings of $\mathsf{T}()$ is used.

To improve readability we omit some technical details of the created PRISM code. This includes, e.g., the code that checks variables range before each assignment (which is required by PRISM). We also omit the description of translation of some parts of the code which is straightforward (e.g. the variables declaration).

[20]For more information about the probabilistic commands in PRISM, please refer to sec. A.

*2) Compiling the program:* The compilation of the ETV program consists of sequential compilation of its parts:

$$\begin{aligned}&\mathsf{T}(\mathsf{comm},\mathsf{contr},\mathsf{scenario}_1,\ldots,\mathsf{scenario}_N)\\&\{\\&\quad\mathsf{T}(\mathsf{comm});\\&\quad\mathsf{T}(\mathsf{contr});\\&\quad\mathsf{T}(\mathsf{scenario}_1);\\&\quad\vdots\\&\quad\mathsf{T}(\mathsf{scenario}_N);\\&\}\end{aligned}$$

In the following sections we define the compilation for each part of the ETV code.

*3) Compiling the communication:* The `communication` section of the ETV code contains the definitions of the communication functions $F_{\mathsf{comm}}=\{f_1,\ldots,f_{|F_{\mathsf{comm}}|}\}$. As a result of the compilation, EthVer creates adequate commands in the communication PRISM module. We denote the function which compiles a communication function by $\mathsf{T}^{\mathsf{comm}}(f)$ and define it below:

$$\begin{aligned}&\mathsf{T}(\mathsf{communication}(f_1,\ldots,f_{|F_{\mathsf{comm}}|}))=\\&\{\\&\quad\mathsf{for}\ f\in\{f_1,\ldots,f_{|F_{\mathsf{comm}}|}\}:\\&\qquad\mathsf{T}^{\mathsf{comm}}(f);\\&\}\end{aligned}$$

$$\begin{aligned}&\mathsf{T}^{\mathsf{comm}}(\mathsf{function}(\textit{fname},[arg_1,\ldots,arg_n],[stm_1,\ldots,stm_k]))=\\&\{\\&\quad\mathsf{for}\ p\in\{1,\ldots,N\}:\\&\qquad\mathsf{comm.add}(\\&\qquad\quad\mathsf{label}:\texttt{communicate\_}||\textit{fname}||p,\\&\qquad\quad\mathsf{from}:1,\mathsf{to}:\sigma_{\mathsf{new}}(),\\&\qquad\quad\mathsf{updates}:[(\texttt{sender},p)]);\\&\qquad\mathsf{comm.add}(\\&\qquad\quad\mathsf{from}:\sigma_{\mathsf{curr}}(),\mathsf{to}:\sigma_{\mathsf{new}}(),\\&\qquad\quad\mathsf{updates}:[(arg_1,arg_1||p),\ldots,(arg_n,arg_n||p)]);\\&\quad\texttt{mod}=\mathsf{comm};\\&\quad\mathsf{for}\ i\in\{1,\ldots,k\}:\\&\qquad\mathsf{T}(stm_i);\\&\quad\mathsf{comm.add}(\mathsf{from}:\sigma_{\mathsf{curr}}(),\mathsf{to}:1);\\&\}\end{aligned}$$

This function creates two special commands in the communication module which are synchronized with the commands from the player module whenever a communication function is being called using `sendCommunication`. It also compiles all the statements of the function and adds an extra command with the transition from the final state of the last statement

to state 1. The mechanism of calling the communication functions will be described in details in sec. D9.

*4) Compiling the contract:* The contract section of the ETV program contains the set of definitions of the contract functions $F_{\mathsf{contr}} = \{f_1, \ldots, f_{|F_{\mathsf{contr}}|}\}$. We model the contract with two PRISM modules: `contract` and `blockchain`, which we describe in details in this section.

The Ethereum transaction can always be assigned to one of the four states:

- NONE,
- BROADCAST,
- EXECUTED,
- INVALIDATED.

The `blockchain` module is responsible for handling the changes of the transaction state. On the other hand, the `contract` module contains the translation of all the statements from the body of the functions. Both modules are created simultaneously during the compilation of the contract, which is defined below:

$$
\begin{aligned}
&\mathsf{T}(\texttt{contract}(f_1, \ldots, f_{|F_{\mathsf{contr}}|})) = \\
&\{ \\
&\quad \textsf{for } p \in \{1, \ldots, N\} : \\
&\quad\quad \textsf{addContractInitCommand}(p); \\
&\quad \textsf{for } f \in \{f_1, \ldots, f_{|F_{\mathsf{contr}}|}\} : \\
&\quad\quad \mathsf{T}^{\mathsf{contr}}(f); \\
&\quad \textsf{addTimestepCommand}(f_1, \ldots, f_{|F_{\mathsf{contr}}|}); \\
&\}
\end{aligned}
$$

The auxiliary functions addContractInitCommand and addTimestepCommand are defined as follows:

$$
\begin{aligned}
&\textsf{addContractInitCommand}(p) = \\
&\{ \\
&\quad \textsf{contr.add}( \\
&\quad\quad \textsf{from} : 0, \textsf{to} : \texttt{next\_state}, \\
&\quad\quad \textsf{guards} : [(\texttt{sender} = p), (\texttt{balance}||p \geq \texttt{value})] \\
&\quad\quad \textsf{updates} : [(\texttt{balance}||p, \texttt{balance}||p - \texttt{value}), \\
&\quad\quad\quad (\texttt{contract\_balance}, \\
&\quad\quad\quad \texttt{contract\_balance} + \texttt{value})]); \\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&\textsf{addTimestepCommand}(f_1, \ldots, f_{|F_{\mathsf{contr}}|}) = \\
&\{ \\
&\quad \textsf{blockchain.add}( \\
&\quad\quad \textsf{label} : \texttt{time\_step}, \\
&\quad\quad \textsf{guards} : \\
&\quad\quad\quad [(\texttt{time\_elapsed} < \texttt{MAX\_TIME}), \\
&\quad\quad\quad (f_1.name||\texttt{\_status}||1 \neq \texttt{T\_BROADCAST}), \ldots, \\
&\quad\quad\quad\quad \ldots, (f_k.name||\texttt{\_status}||1 \neq \texttt{T\_BROADCAST}), \\
&\quad\quad\quad \vdots \\
&\quad\quad\quad (f_1.name||\texttt{\_status}||N \neq \texttt{T\_BROADCAST}), \ldots, \\
&\quad\quad\quad\quad \ldots, (f_k.name||\texttt{\_status}||N \neq \texttt{T\_BROADCAST})], \\
&\quad\quad \textsf{updates} : [(\texttt{time\_elapsed}, \texttt{time\_elapsed} + 1)]); \\
&\}
\end{aligned}
$$

The first function (addContractInitCommand) creates a special command in the contract module which is executed before the execution of each contract function. This command is needed to check if the sender of the transaction has sufficient funds and to properly pass the sender address to the function call. The second function (addTimestampCommand) adds a command to the blockchain module. This command allows to increase the `time_elapsed` variable in case no transaction is waiting in the `T_BROADCAST` state. This enables the passage of time and models the requirement, that as long as there are waiting transactions in the `T_BROADCAST` state, they eventually will be executed by the network.

The main part of the compilation of the contract is the compilation of all of its functions. The compilation of a function consists of creating some auxiliary commands related to this function (we describe this part separately in the addBlockchainCommands and addContractExecCommand functions) and compiling all the statements of the function. After the last statement, an extra command is added with the transition to the first state. Hence, at the end of every execution of the function, the contract module always lands in state 1. Below we describe in details, how a particular contract function is compiled using $\mathsf{T}^{\mathsf{contr}}$ function:

$$
\begin{aligned}
&\mathsf{T}^{\mathsf{contr}}(\textsf{function}(\mathit{fname}, [\mathit{arg}_1, \ldots, \mathit{arg}_n], [\mathit{stm}_1, \ldots, \mathit{stm}_k])) = \\
&\{ \\
&\quad \textsf{for } p \in \{1, \ldots, N\} : \\
&\quad\quad \textsf{addBlockchainCommands}(f, p); \\
&\quad\quad \textsf{addContractExecCommand}(f, p); \\
&\quad \texttt{mod} = \textsf{contr}; \\
&\quad \textsf{for } i \in \{1, \ldots, k\} : \\
&\quad\quad \mathsf{T}(\mathit{stm}_i); \\
&\quad \textsf{contr.add}(\textsf{from} : \sigma_{\mathsf{curr}}(), \textsf{to} : 1); \\
&\}
\end{aligned}
$$

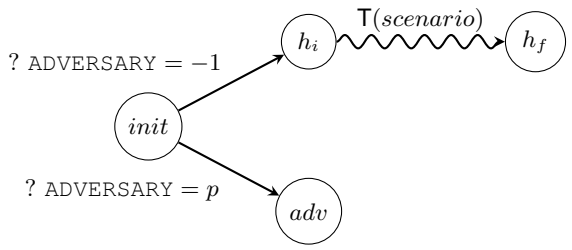
The mechanism of calling the contract functions is slightly more complex than the analogous mechanism in the communication module. In addition to the compilation of all the function statements, two auxiliary functions addBlockchainCommands addContractExecCommand are executed. We define these functions below:

addBlockchainCommands($f =$ function(
$\mathit{fname}, [\mathit{arg}_1, \ldots, \mathit{arg}_n], [\mathit{stm}_1, \ldots, \mathit{stm}_k]), p) =$
{
  blockchain.add(
    label : broadcast_||*fname*||$p$,
    guards : [(contrstate $= 1$), (commstate $= 1$),
      (*fname*||_status||$p \neq$ T_BROADCAST)],
    updates : [(*fname*||_status||$p$, T_BROADCAST)]);
  if $f$ is payable
  then
    blockchain.add(
      label : broadcast_||*fname*,
      guards : [(contrstate $= 1$), (commstate $= 1$),
        (*fname*||_status||$p =$ T_BROADCAST),
        (*fname*||_value||$p \leq$ balance||$p$)],
      updates : [(sender, $p$),
        (*fname*||_status||$p$, T_EXECUTED),
        (value,*fname*||_value||$p$),
        $(arg_1, arg_1||p), \ldots, (arg_n, arg_n||p)$]);
    blockchain.add(
      guards : [(contrstate $= 1$), (commstate $= 1$),
        (*fname*||_status||$p =$ T_BROADCAST),
        (*fname*||_value||$p >$ balance||$p$)],
      updates :
        [(*fname*||_status||$p$, T_INVALIDATED)]);
  else
    blockchain.add(
      label : broadcast_||*fname*,
      guards : [(contrstate $= 1$), (commstate $= 1$),
        (*fname*||_status||$p =$ T_BROADCAST)],
      updates : [(sender, $p$),
        (*fname*||_status||$p$, T_EXECUTED),
        $(arg_1, arg_1||p), \ldots, (arg_n, arg_n||p)$]);
}

addContractExecCommand$(f, p) =$
{
  contr.add(
    label : broadcast_||*fname*,
    guards : [(contrstate $= 1$)],
    updates : [(contrstate, 0), (next_state, $\sigma_{\mathsf{new}}()$)]);
}

The commands created by these functions are synchronized with the corresponding commands from the player module whenever a contract function is called using sendTransaction. The exact mechanism of calling the contract functions will be described in details in sec. D10.

*5) The player modules:* As it was already described in sec. VI, the player module consists of two parts. The first part corresponds to the honest execution of the scenario (ADVERSARY $= -1$ mode) and is the result of the translation of the scenario T($scenario$). The second part of the module models the adversarial behavior of the player $p$ (ADVERSARY $= p$ mode). The schematic view of the player $p$ module is depicted below:




So the translation of a scenario consists of two separate functions, each creating a separate part of the model:

T(scenario($[\mathit{stm}_1, \ldots, \mathit{stm}_k]$)) =
{
  $\mathsf{T}^{\mathsf{honest}}$(scenario);
  for $i \in \{1, \ldots, |F_{\mathsf{comm}}|\}$ :
    addAdversarialCommCommands($f_i$);
  for $i \in \{1, \ldots, |F_{\mathsf{contr}}|\}$ :
    addAdversarialContrCommands($f_i$);
}

Below we define the both parts of the translation.

*a) The honest part:* The honest part of the player module is created by translating the scenario part of the ETV code. We defined the operational semantics of a scenario as follows:

$$\overline{[[\mathsf{scenario}(\mathit{name}, \mathit{decls}, \mathit{stms})]]_\rho = [[\mathit{stms}]]_\rho}$$

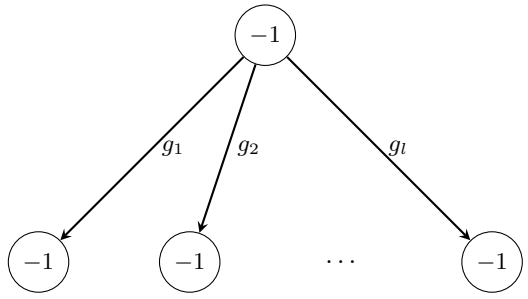
Now we define the translation of the scenario into the `player` module.

$$\begin{aligned}&\mathsf{T}^{\mathsf{honest}}(\mathsf{scenario}([stm_1,\ldots,stm_k])) =\\ &\{\\ &\quad \mathsf{for}\ i \in \{1,\ldots,k\}:\\ &\qquad \mathsf{T}(stm_i);\\ &\}\end{aligned}$$

**Theorem 1.** *The translation of the scenario defined as above preserves its operational semantics.*

*Proof.* The model created by the translation of a scenario looks as in the fig. 1. Any execution of the scenario corresponds to some path in this graph. Hence, if the semantics of each statement is preserved by the translation, then, by induction, the semantics of the whole execution is also preserved. In the following sections we will define the translation for every statement of the ETV language and prove that the translation preserves the semantics of the statements. That will imply that the semantics of the whole scenario is also preserved. □

*b) The adversarial part:* The adversarial part of the `player` module consists of only one adversarial state $(-1)$ and commands $g_1,\ldots,g_l$ going from state $(-1)$ back to the same state:




The commands $g_1,\ldots,g_l$ are the commands generated with addAdversarialCommCommands for every communication function and the commands generated with addAdversarialContrCommands for every communication function.

The advAdversarialCommCommands auxiliary function generates a separate `sendCommunication` command for every possible valuation of the arguments of the function:

$$\begin{aligned}&\mathsf{addAdversarialCommCommands}(\mathsf{function}(\\ &\qquad fname,[arg_1,\ldots,arg_n],[stm_1,\ldots,stm_k])) =\\ &\{\\ &\quad \mathsf{for}\ (x_1,\ldots,x_n \in valuations(arg_1,\ldots,arg_n):\\ &\qquad \mathsf{T}^{\mathsf{adv}}(f.\texttt{sendCommunication}(x_1,\ldots,x_n);\\ &\}\end{aligned}$$

The $\mathsf{T}^{\mathsf{adv}}(f.\texttt{sendCommunication})$ function creates the PRISM commands identical to those created by $\mathsf{T}^{\mathsf{honest}}(f.\texttt{sendCommunication})$ with the only difference that instead of transitions from state $\sigma_{\mathsf{curr}}()$ to $\sigma_{\mathsf{new}}()$ it creates all the commands from state $(-1)$ to state $(-1)$.

Similarly, the advAdversarialContrCommands(f) function generates a separate `sendTransaction` command for every possible valuation of the arguments and every possible *value* of the call:

$$\begin{aligned}&\mathsf{addAdversarialContrCommands}(\mathsf{function}(\\ &\qquad fname,[arg_1,\ldots,arg_n],[stm_1,\ldots,stm_k])) =\\ &\{\\ &\quad \mathsf{for}\ (x_1,\ldots,x_n,u) \in valuations(arg_1,\ldots,arg_n,value):\\ &\qquad \mathsf{T}^{\mathsf{adv}}(f.\texttt{sendTransaction}\,(x_1,\ldots,x_n,\\ &\qquad\quad \texttt{\{value:}u\texttt{\}}));\\ &\}\end{aligned}$$

*6) Compiling basic statements and expressions:* In this section we define the translation of the statements and expressions that are typical to every imperative language like C or Java.

*a) Expressions:* The translation of basic expressions is straightforward, however we define it here for completeness:

$$\begin{aligned}&\mathsf{T}(\texttt{true}) = \texttt{true}\\ &\mathsf{T}(\texttt{false}) = \texttt{false}\\ &\mathsf{T}(\texttt{finney}(E)) = E\\ &\mathsf{T}(\texttt{this}) = \texttt{sender}\\ &\mathsf{T}(\texttt{(}E\texttt{)}) = (E)\\ &\mathsf{T}(\texttt{msg.value}) = \texttt{value}\\ &\mathsf{T}(\texttt{msg.sender}) = \texttt{sender}\\ &\mathsf{T}(E) = E \quad \mathsf{for}\ E \in \{\langle String\rangle,\langle Integer\rangle,\langle Ident\rangle\}\\ &\mathsf{T}(I[E]) = \{\\ &\quad v = \mathsf{T}(E);\\ &\quad \mathsf{return}\ I||v;\\ &\}\end{aligned}$$

*b) The assignment:* We define the translation of the assignment statement as follows:

$$\begin{aligned}&\mathsf{T}(I = E) =\\ &\{\\ &\quad v = \mathsf{T}(E);\\ &\quad \texttt{mod.}\mathsf{add}(\\ &\qquad \mathsf{from} : \sigma_{\mathsf{curr}}(), \mathsf{to} : \sigma_{\mathsf{new}}(),\\ &\qquad \mathsf{updates} : [(I,E)]);\\ &\}\end{aligned}$$

This code first evaluates the expression $E$ and then it assigns the result to the $I$ variable, so it clearly satisfies the semantics that we defined before:

$$\frac{I = \langle Ident\rangle \quad [[E]]_\rho = v}{[[I = E]]_\rho = \rho.V\{I/k\}}$$

*c) The array assignment:* The PRISM language does not support arrays. Therefore we implement arrays of constant size


Fig. 1. Translation of a scenario

as the set of variables with the name equal to the concatenation of the name of array and the index:

$$
\begin{aligned}
&\mathsf{T}(I[E_1] = E_2) = \\
&\{ \\
&\quad i = \mathsf{T}(E_1); \\
&\quad v = \mathsf{T}(E_2); \\
&\quad \texttt{mod}.\mathsf{add}( \\
&\qquad \mathsf{from} : \sigma_{\mathsf{curr}}(), \mathsf{to} : \sigma_{\mathsf{new}}(), \\
&\qquad \mathsf{updates} : [(I||i, v)]); \\
&\}
\end{aligned}
$$

Clearly, this model also satisfies the semantics of the semantics of the array assignment from the ETV language:

$$\frac{I = \langle \mathit{Ident} \rangle \quad [[E_1]]_\rho = i \quad [[E_2]]_\rho = v}{[[I[E_1] = E_2]]_\rho = \rho.V\{v/(I||i)\}}$$

*d) The `if` statement:* Recall that we defined the semantics of the `if` statement using the following rules:

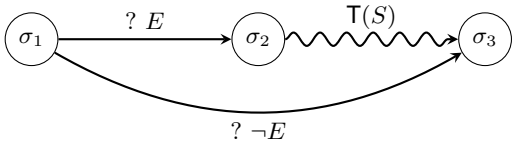
$$\frac{[[E]]_\rho = \texttt{true}}{[[\texttt{if } (E) \ S]]_\rho = [[S]]_\rho} \tag{1}$$

$$\frac{[[E]]_\rho = \texttt{false}}{[[\texttt{if } (E) \ S]]_\rho = \rho} \tag{2}$$

We now can define the translation of the `if` statement:

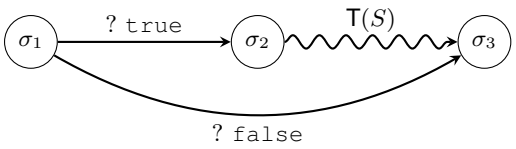
$$
\begin{aligned}
&\mathsf{T}(\texttt{if } (E) \ S) = \\
&\{ \\
&\quad v = \mathsf{T}(E); \\
&\quad \sigma_1 = \sigma_{\mathsf{curr}}(); \\
&\quad \texttt{mod}.\mathsf{add}(\mathsf{from} : \sigma_{\mathsf{curr}}(), \mathsf{to} : \sigma_{\mathsf{new}}(), \mathsf{guards} : v); \\
&\quad \sigma_2 = \sigma_{\mathsf{curr}}(); \\
&\quad \mathsf{T}(S); \\
&\quad \sigma_3 = \sigma_{\mathsf{curr}}(); \\
&\quad \texttt{mod}.\mathsf{add}(\mathsf{from} : \sigma_1, \mathsf{to} : \sigma_3, \mathsf{guards} : \neg v); \\
&\}
\end{aligned}
$$

The PRISM model obtained from this translation is depicted below:

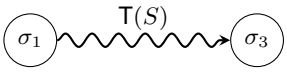



**Lemma 1.** *The translated `if` statement preserves its operational semantics*

*Proof.* Let us start with the case when $E$ evaluates to `true` and prove that in this case the obtained model preserves (1). When $E$ evaluates to `true`, the generated model looks as follows:




which is equivalent to:

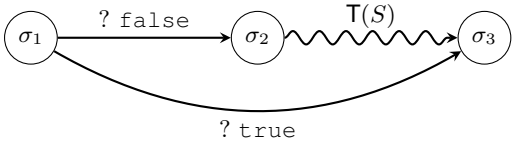



which is a model generated by $\mathsf{T}(S)$. Hence, indeed:

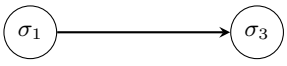
$$[[\mathsf{T}(\texttt{if } (E) \ S)]]_\rho = [[\mathsf{T}(S)]]_\rho$$

so (1) is preserved.

Now, suppose the opposite, i.e., $E$ evaluates to `false`. Then the generated model will look as follows:




which is equivalent to:




so in this case

$$[[\mathsf{T}(\texttt{if } (E) \ S)]]_\rho = \rho$$

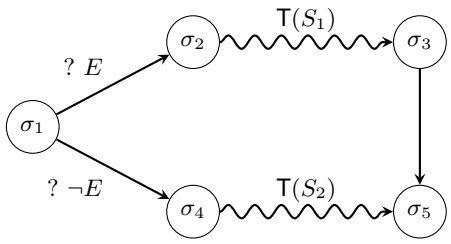
so (2) is also preserved, which ends the proof. □

*e) The `if-else` statement:* Similarly we define the translation of the `if-else` statement:

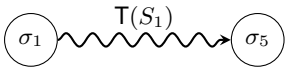
$$
\begin{aligned}
&\mathsf{T}(\texttt{if } (E)\ S_1 \texttt{ else } S_2) = \\
&\{ \\
&\quad v = \mathsf{T}(E); \\
&\quad \sigma_1 = \sigma_{\mathsf{curr}}(); \\
&\quad \texttt{mod}.\mathsf{add}(\mathsf{from} : \sigma_{\mathsf{curr}}(), \mathsf{to} : \sigma_{\mathsf{new}}(), \mathsf{guards} : v); \\
&\quad \sigma_2 = \sigma_{\mathsf{curr}}(); \\
&\quad \mathsf{T}(S_1); \\
&\quad \sigma_3 = \sigma_{\mathsf{curr}}(); \\
&\quad \texttt{mod}.\mathsf{add}(\mathsf{from} : \sigma_1, \mathsf{to} : \sigma_{\mathsf{new}}(), \mathsf{guards} : \neg v); \\
&\quad \sigma_4 = \sigma_{\mathsf{curr}}(); \\
&\quad \mathsf{T}(S_2); \\
&\quad \sigma_5 = \sigma_{\mathsf{curr}}(); \\
&\quad \texttt{mod}.\mathsf{add}(\mathsf{from} : \sigma_3, \mathsf{to} : \sigma_5); \\
&\}
\end{aligned}
$$

**Lemma 2.** *The model generated by the described translation preserves the semantics of the `if-else` statement from the ETV language.*

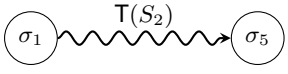
*Proof.* The generated model can be presented as the following graph:




If $E$ evaluates to true, the graph can be reduced to:




If $E$ evaluates to false, the graph can be reduced to:




Therefore, the generated model satisfies the semantics of the `if-else` statement:

$$\frac{[[E]]_\rho = b \in \{\texttt{true}, \texttt{false}\}}{[[\texttt{if } (E)\ S_{\texttt{true}} \texttt{ else } S_{\texttt{false}}]]_\rho = [[S_b]]_\rho}$$

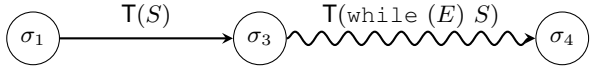
□

*f) The `while` loop:* We define the translation of the `while` loop as follows:

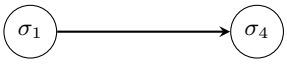
$$
\begin{aligned}
&\mathsf{T}(\texttt{while } (E)\ S) = \\
&\{ \\
&\quad v = \mathsf{T}(E); \\
&\quad \sigma_1 = \sigma_{\mathsf{curr}}(); \\
&\quad \texttt{mod}.\mathsf{add}(\mathsf{from} : \sigma_{\mathsf{curr}}(), \mathsf{to} : \sigma_{\mathsf{new}}(), \mathsf{guards} : v); \\
&\quad \sigma_2 = \sigma_{\mathsf{curr}}(); \\
&\quad \mathsf{T}(\texttt{while } (E)\ S); \\
&\quad \sigma_3 = \sigma_{\mathsf{curr}}(); \\
&\quad \texttt{mod}.\mathsf{add}(\mathsf{from} : \sigma_1, \mathsf{to} : \sigma_3, \mathsf{guards} : \neg v); \\
&\}
\end{aligned}
$$

**Lemma 3.** *The translation defined above preserves the semantics of the `while` loop.*

*Proof.* The model generated by the translation can be presented as the graph in fig. 2: If $E$ evaluates to true, then the graph can be reduced to:




If $E$ evaluates to false, then the graph can be reduced to:




Therefore the model satisfies the ETV semantics of the `while` loop:

$$\frac{[[E]]_\rho = \texttt{true} \quad [[S]]_\rho = \rho'}{[[\texttt{while } (E)\ S]]_\rho = [[\texttt{while } (E)\ S]]_{\rho'}}$$
$$\frac{[[E]]_\rho = \texttt{false}}{[[\texttt{while } (E)\ S]]_\rho = \rho}$$

□

*7) The `wait` statement:* `wait` is a special statement introduced in ETV to indicate the place in the scenario at which the player should interrupt the execution of the protocol and wait until one of the two conditions holds:

- the given expression evaluates to true,
- the given time period has elapsed.

We define the translation of the `wait` statement below:

$$
\begin{aligned}
&\mathsf{T}(\texttt{wait}(E_1, E_2)) = \\
&\{ \\
&\quad v_1 = \mathsf{T}(E_1); \\
&\quad v_2 = \mathsf{T}(E_2); \\
&\quad \sigma_1 = \sigma_{\mathsf{curr}}(); \\
&\quad \texttt{mod}.\mathsf{add}(\mathsf{from} : \sigma_{\mathsf{curr}}(), \mathsf{to} : \sigma_{\mathsf{new}}(), \mathsf{guards} : v_1); \\
&\quad \sigma_2 = \sigma_{\mathsf{curr}}(); \\
&\quad \texttt{mod}.\mathsf{add}(\mathsf{from} : \sigma_1, \mathsf{to} : \sigma_2, \\
&\qquad \mathsf{guards} : \texttt{time\_elapsed} \geq v_2); \\
&\}
\end{aligned}
$$

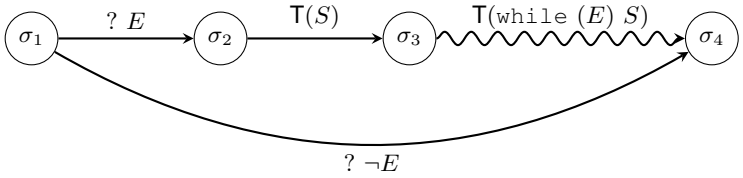


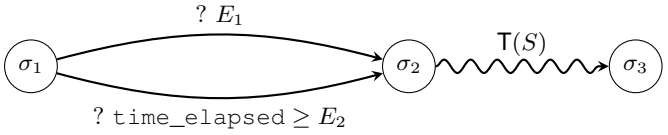

Fig. 2. Translation of the `while` statement

Recall that we defined the semantics of `wait` using the statement $S$ which occurs after the `wait` statement in the scenario:

$$\frac{\frac{[[E_1]]_\rho=\texttt{true}}{[[\texttt{wait}(E_1,E_2);S]]_\rho=[[S]]_\rho}}{\frac{\texttt{<time>}\geq[[E_2]]_\rho}{[[\texttt{wait}(E_1,E_2);S]]_\rho=[[S]]_\rho}} \tag{3}$$

**Lemma 4.** *The PRISM code generated with the commands described above satisfies the semantics (3) of the* `wait` *statement.*

*Proof.* Let us draw a graph that represents the translation of a `wait` statement followed by the $S$ statement:




If none of the guards is satisfied ($E_1 = \texttt{false}$ and $\texttt{time\_elapsed} < E_2$), then no transition from state $\sigma_1$ is possible and hence the model is blocked in state $\sigma_1$. On the other hand, if any of the guards is satisfied, then the transition to $\sigma_2$ is active and the commands generated by the translation $\mathsf{T}(S)$ can be executed. Hence, the semantics (3) is satisfied. □

*8) The* `transfer` *method:* The `transfer` method is used to send money from contract to users. We defined the semantics of this method as follows[21]:

$$\frac{u \leq \rho.U(\mathcal{C})}{[[\mathcal{A}.\texttt{transfer}(u)]]_\rho = \rho - \mathcal{C} : u + \mathcal{A} : u} \tag{4}$$

$$\frac{u > \rho.U(\mathcal{C})}{[[\mathcal{A}.\texttt{transfer}(u)]]_\rho = \rho} \tag{5}$$

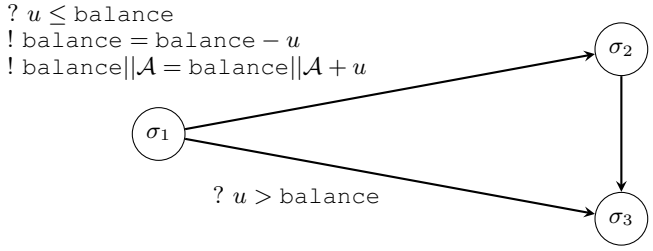
[21]Recall that we introduced the notation $\rho - \mathcal{X} : u + \mathcal{Y} : u$ to indicate the decrease of the balance of $\mathcal{X}$ and the increase of the balance of $\mathcal{Y}$ by $u$, where $\mathcal{C}$ denotes the contract balance.

Now we can define the translation of the `transfer` statement to PRISM:

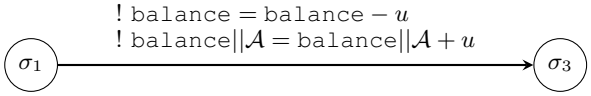
$$\begin{aligned}
&\mathsf{T}(\mathcal{A}.\texttt{transfer}(u)) = \\
&\{ \\
&\quad \sigma_1 = \sigma_{\mathsf{curr}}(); \\
&\quad \texttt{mod}.\mathsf{add}( \\
&\qquad \mathsf{from} : \sigma_{\mathsf{curr}}(),\ \mathsf{to} : \sigma_{\mathsf{new}}(), \\
&\qquad \mathsf{guards} : [(u \leq \texttt{balance})], \\
&\qquad \mathsf{updates} : [(\texttt{balance}, \texttt{balance} - u), \\
&\qquad\quad (\texttt{balance}||\mathcal{A}, \texttt{balance}||\mathcal{A} + u)]); \\
&\quad \sigma_2 = \sigma_{\mathsf{curr}}(); \\
&\quad \texttt{mod}.\mathsf{add}( \\
&\qquad \mathsf{from} : \sigma_1,\ \mathsf{to} : \sigma_{\mathsf{new}}(), \\
&\qquad \mathsf{guards} : [(u > \texttt{balance})]); \\
&\quad \sigma_3 = \sigma_{\mathsf{curr}}(); \\
&\quad \texttt{mod}.\mathsf{add}( \\
&\qquad \mathsf{from} : \sigma_2,\ \mathsf{to} : \sigma_3); \\
&\}
\end{aligned}$$

**Lemma 5.** *The translated* `transfer` *method preserves its operational semantics*

*Proof.* The PRISM model created with the translation looks as follows:

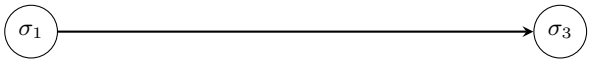



If $u \leq \texttt{balance}$ then the model is equivalent to:




so it preserves (4).

On the other hand, if $u > \texttt{balance}$, then the created model is equivalent to:



Thus, it preserves (5). □

*9) The* `sendCommunication` *method:* The `sendCommunication` method is used to execute a communication function from the scenario. We have defined the following semantics of this statement:

$$\frac{\begin{array}{c}[[E_1,\ldots,E_n]]_\rho = (v_1,\ldots,v_n)\\ f(arg_1,\ldots,arg_n)\{S\} \in F_{\mathsf{comm}}\\ \rho' = \rho.V\{v_1/arg_1,\ldots,v_n/arg_n\}\end{array}}{[[f.\texttt{sendCommunication}(E_1,\ldots,E_n)]]_\rho = [[S]]_{\rho'}} \quad (6)$$

We now define the translation of `sendCommunication` from ETV to PRISM:

$$\begin{array}{l}\mathsf{T}(f.\texttt{sendCommunication}(E_1,\ldots,E_n)) =\\ \{\\ \quad v_1 = \mathsf{T}(E_1);\\ \quad \vdots\\ \quad v_n = \mathsf{T}(E_n);\\ \quad \texttt{mod}.\mathsf{add}(\\ \quad\quad \mathsf{from} : \sigma_{\mathsf{curr}}(),\ \mathsf{to} : \sigma_{\mathsf{new}}(),\\ \quad\quad \mathsf{updates} : [(arg_1||p, v_1),\ldots,(arg_n||p, v_n)]);\\ \quad \texttt{mod}.\mathsf{add}(\\ \quad\quad \mathsf{label} : \texttt{communicate\_}||\textit{fname}||p,\\ \quad\quad \mathsf{from} : \sigma_{\mathsf{curr}}(),\ \mathsf{to} : \sigma_{\mathsf{new}}());\\ \}\end{array}$$

**Lemma 6.** *The model created by the translation described above preserves the semantics of* `sendCommunication`*.*

*Proof.* The label of the last command (`communicate_`$||$*fname*$||p$) is the same as the label of the command created in the `communication` module during the compilation $\mathsf{T}^{\mathsf{comm}}(f)$. Hence, the two commands are *synchronized* using the PRISM synchronization mechanism. In the figure 3 the commands $\pi_3 \to \pi_4$ and $\sigma_1 \to \sigma_2$ are synchronized. Before these commands the values $v_1,\ldots,v_n$ are computed in the commands between states $\pi_1$ and $\pi_2$ and then they are assigned to variables $arg_1||p,\ldots,arg_n||p$ in command $\pi_2 \to \pi_3$. Then, after the synchronized command, the computed values are assigned to the variables $arg_1,\ldots,arg_n$, which are directly accessible inside the `communication` module while executing commands from $\mathsf{T}$(*stms*).

Furthermore, we are guaranteed that the $(arg_1,\ldots,arg_n)$ variables will be not changed in the meantime by any parallel call of `sendCommunication`: in order to execute the synchronized command, the value of `commstate` variable must be equal to 1. Once this command is executed, `commstate` is set to other values until all the statements are executed, and then it is set back to 1. Therefore, between states $\sigma_3$ and $\sigma_4$ all the statements of the function are executed, with the correct values of the function arguments, so by induction, the semantics (6) is preserved. □

*10) The* `sendTransaction` *method:* The `sendTransaction` method is used to execute the contract function. We defined the semantics of this statement as follows:

$$\frac{\begin{array}{l}[[E_1,\ldots,E_n]]_\rho = (v_1,\ldots,v_n) \quad \rho.V(\texttt{sender}) = \mathcal{A}\\ [[E]]_\rho = u \le \rho.U(\mathcal{A}) \quad f(arg_1,\ldots,arg_n)\{S\} \in F_{\mathsf{contr}}\\ \rho' = \rho - \mathcal{A} : u + \mathcal{C} : u\\ \rho'' = \rho'.V\{u/\texttt{value}, v_1/arg_1,\ldots,v_n/arg_n\}\end{array}}{\begin{array}{l}[[f.\texttt{sendTransaction}(E_1,\ldots,E_n,\{\texttt{value} : E\})]]_\rho\\ = [[S]]_{\rho''}\end{array}} \quad (7)$$

$$\frac{\rho.V(\texttt{sender}) = \mathcal{A} \quad [[E]]_\rho > \rho.U(\mathcal{A})}{[[f.\texttt{sendTransaction}(E_1,\ldots,E_n,\{\texttt{value} : E\})]]_\rho = \rho} \quad (8)$$

We define the following translation of the `sendTransaction` method to PRISM:

$$\begin{array}{l}\mathsf{T}(f.\texttt{sendTransaction}(E_1,\ldots,E_n,\{\texttt{value} : E\})) =\\ \{\\ \quad v_1 = \mathsf{T}(E_1);\\ \quad \vdots\\ \quad v_n = \mathsf{T}(E_n);\\ \quad u = \mathsf{T}(E);\\ \quad \texttt{mod}.\mathsf{add}(\\ \quad\quad \mathsf{label} : \texttt{broadcast\_}||\textit{fname}||p,\\ \quad\quad \mathsf{from} : \sigma_{\mathsf{curr}}(),\ \mathsf{to} : \sigma_{\mathsf{new}}(),\\ \quad\quad \mathsf{updates} : [(arg_1||p, v_1),\ldots,(arg_n||p, v_n),\\ \quad\quad\quad (\textit{fname}||\texttt{\_value}||p, u)]);\\ \}\end{array}$$

**Lemma 7.** *The model created by the translation described above preserves the semantics of* `sendTransaction`*.*

*Proof.* The label (`broadcast_`$||$*fname*$||p$) of the created command is the same as the label of the corresponding command created in the `blockchain` module during the compilation $\mathsf{T}^{\mathsf{contr}}(f)$. Furthermore, the command labelled (`broadcast_`$||$*fname*) from the `blockchain` module is synchronized with the corresponding command from thet `contract` module. The whole picture of the 3 modules is presented in fig. 4. The execution of `sendTransaction` call is performed as follows:

1) The expressions for the values of the arguments are evaluated (command $\pi_1 \to \pi_2$).
2) If the function has not yet been broadcast by player $p$, then the values of the arguments are assigned to $(arg_1||p,\ldots,arg_n||p)$ and the status of the function in set to `T_BROADCAST` (synchronized commands $\pi_2 \to \pi_3$ and $\beta_1 \to \beta_2$).
3) If the balance of player $p$ is sufficient, then the synchronized commands $\beta_2 \to \beta_3$ and $\sigma_1 \to \sigma_2$ are executed. The proper values of `sender`, $(arg_1,\ldots,arg_n)$ and

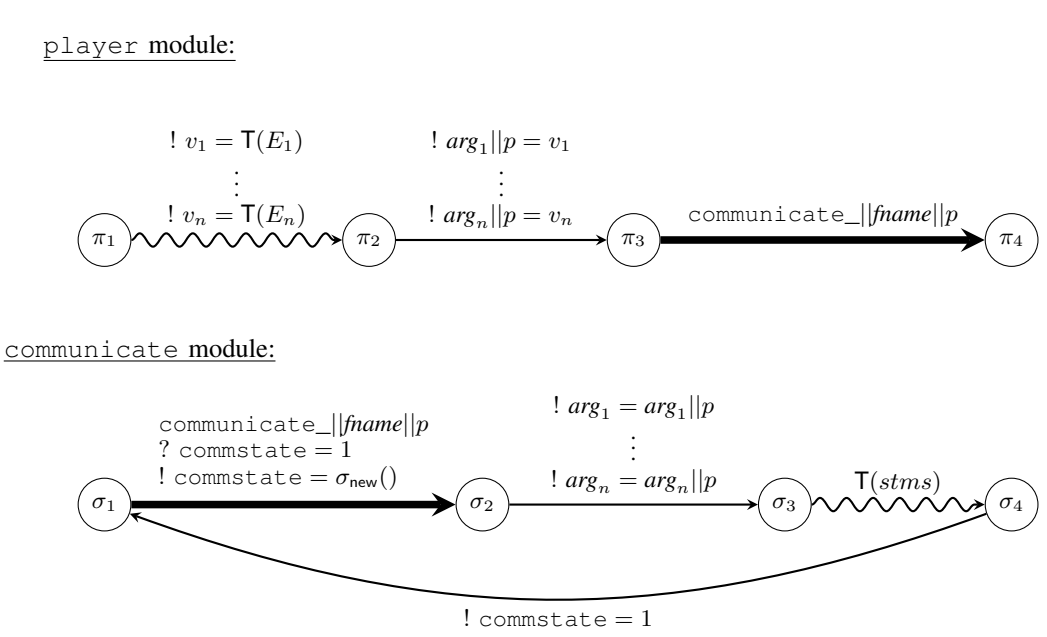



Fig. 3. Translation of `sendCommunication` statement

`value` are set. If the balance is not sufficient, then the $\beta_2 \rightarrow \beta_4$ command is executed, the status of the transaction is set to `T_INVALIDATED` and the execution is over.

4) If the status has been successfully changed to `T_EXECUTED`, then in the $\sigma_2 \rightarrow \sigma_3$ command the `sender` variable is set and the balances are updated.
5) Between states $\sigma_3$ and $\sigma_4$ all the statements of the function are executed.

Note that `sendTransaction` can be called multiple times before the other transactions are executed by the blockchain, so there may be multiple transactions in `T_BROADCAST` state at the same time. This reflects the behavior of the real Ethereum blockchain, where users can call any contract function many times in parallel, but the actual execution of the contract is made sequentially when the miner mines a new block.

In our model, only one contract function can be executed at the same time due to the `contrstate` variable: in order to execute the synchronized $(\beta_2 \rightarrow \beta_3, \sigma_1 \rightarrow \sigma_2)$ commands, `contrstate` must be equal to 1. Whenever some transaction is taken to be executed, `contrstate` is changed to 0, and then to other values while executing the function statements. It is set back to 1, once all the statements are successfully executed. So no other transaction can be executed in parallel to the ongoing execution.

Hence, as long as the semantics of each function statement is preserved, by induction, the semantics of the whole function execution (7) and (8) is also preserved. □

*11) The `random` function:* The `random(R)` function is designed to return a number from $\{0, 1, \ldots, (R-1)\}$ set with uniform probability. We defined the semantics of this function as follows:

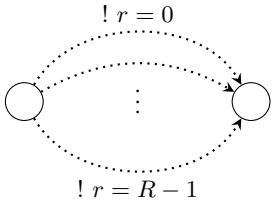
$$[[\texttt{random}(R)]]_\rho = \begin{cases} 1/R : 0 \\ \vdots \\ 1/R : (R-1) \end{cases} \tag{9}$$

To simulate the `random` function in the PRISM model we need to create a temporary local variable. We will use the special $var_{\mathsf{new}}()$ function to return the index of the first unused local variable, similarly to $\sigma_{\mathsf{new}}()$. Now we are ready to define the translation of the `random` function:

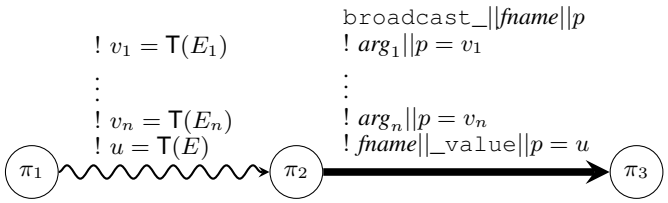
$\mathsf{T}(\texttt{random}(R)) =$
{
  $r = var_{\mathsf{new}}();$
  mod.add(
    from : $\sigma_{\mathsf{curr}}()$, to : $\sigma_{\mathsf{new}}()$,
    updates : $\{1/R : [(r, 0)], \ldots, 1/R : [(r, R-1)]\}$);
  return $r$;
}

**Lemma 8.** *The model created by the function defined above preserves (9).*

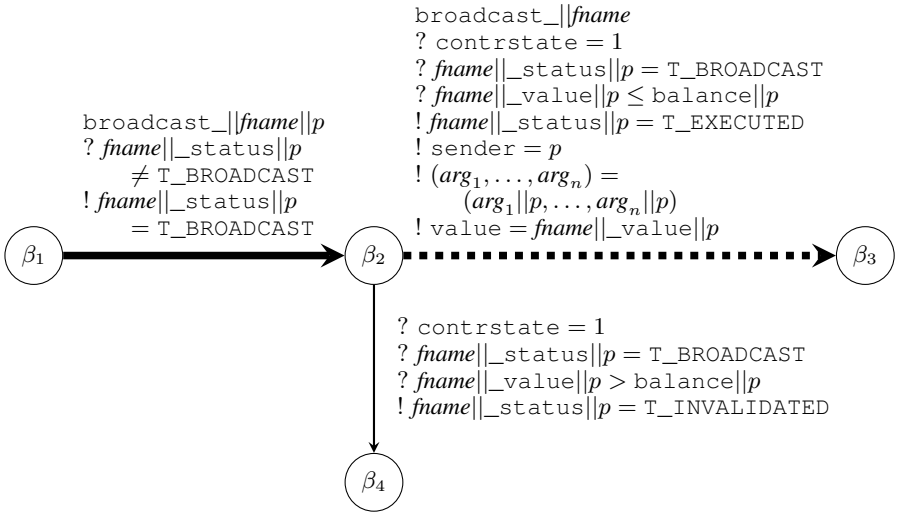
*Proof.* The model created by the $\mathsf{T}(\texttt{random}(R))$ function looks as follows:



`player` module:




`blockchain` module:




`contract` module:

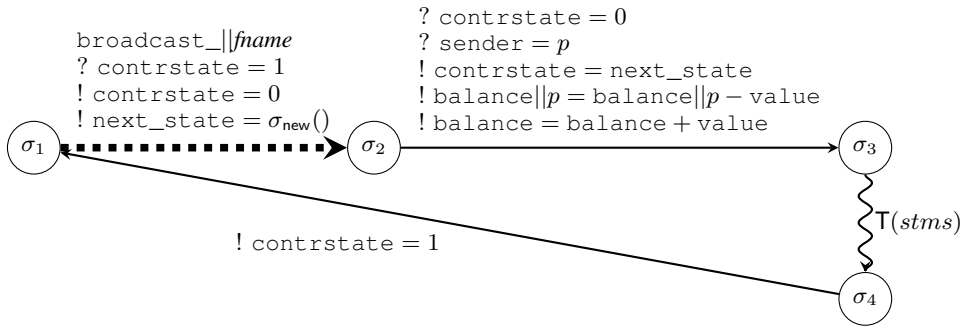



Fig. 4. Translation of `sendTransaction` statement

Each of the transitions is taken with the probability $1/R$. Hence, the final value returned by the expression is distributed as in (9). □

*12) Compiling the cryptographic commitments:* In this section we provide the translation of the commitment operations. Below is the list of ETV statements and expressions related to commitments together with their operational semantics.

$$\frac{\rho.C(c).\mathsf{state} = \mathsf{init}}{[\![c.\texttt{randomCmt()}]\!]_\rho = \begin{cases} 1/R : \rho.C\{0/c.\mathsf{value}, \mathsf{committed}/c.\mathsf{state}, \\ \qquad \mathsf{genHash}(c)/c.\mathsf{hash}\} \\ \vdots \\ 1/R : \rho.C\{(R-1)/c.\mathsf{value}, \mathsf{committed}/c.\mathsf{state}, \\ \qquad \mathsf{genHash}(c)/c.\mathsf{hash}\} \end{cases}} \tag{10}$$

$$\frac{\rho.C(c).\mathsf{state} = \mathsf{committed}}{[[c.\texttt{revealCmt()}]]_\rho = \rho.C\{\mathsf{revealed}/c.\mathsf{state}\}} \quad (11)$$

$$\frac{\rho.C(c).\mathsf{state} = \mathsf{revealed}}{[[\texttt{valueOf}(c)]] = \rho.C(c).\mathsf{value}} \quad (12)$$

$$\frac{\rho.C(c).\mathsf{state} \in \{\mathsf{committed}, \mathsf{revealed}\}}{[[\texttt{hashOf}(c)]] = \rho.C(c).\mathsf{hash}} \quad (13)$$

$$\frac{\rho.C(c).\mathsf{state} \in \{\mathsf{committed}, \mathsf{revealed}\}}{[[\texttt{verCmt}(c,h)]] = (\rho.C(c).\mathsf{hash} = h)} \quad (14)$$

As it was described in sec. VI-D, in PRISM we model a commitment as the object which can be in one of the three states:

- init
- committed
- revealed

The values and the states of all the commitments are stored in a global list allCmt. Whenever a new commitment is created, a new index in this list is reserved for it using $cmt_{\mathsf{new}}()$ function (similarly to $\sigma_{\mathsf{new}}()$ and $var_{\mathsf{new}}()$). We will refer to the index, the state and the value of the commitment $c$ by $c$, $\mathsf{allCmt}(c).\mathsf{state}$ and $\mathsf{allCmt}(c).\mathsf{value}$ respectively.

Below we present the two commitment-related parts of the model generated by *EthVer* — the first part is the direct result of the compilation of the contract and the scenarios, while the second part models the adversarial usage of the commitments.

*a) The honest usage of commitments:* Using the syntax described above, we define the translation of the `randomCmt` method as follows:

$$\begin{array}{l}
\mathsf{T}(c.\texttt{randomCmt()}) = \\
\{ \\
\quad \texttt{mod}.\mathsf{add}( \\
\quad\quad \mathsf{from} : \sigma_{\mathsf{curr}}(), \mathsf{to} : \sigma_{\mathsf{new}}(), \\
\quad\quad \mathsf{updates} : [(c, cmt_{\mathsf{new}}()), (\mathsf{allCmt}(c).\mathsf{state}, \mathsf{committed})]); \\
\}
\end{array}$$

The $cmt_{\mathsf{new}}()$ function creates a new commitment variable and returns its index in the allCmt list. Hence, the commitments can be tracked by the index in the list and this index plays the role of the commitment hash. The only way to change the commitment value after it is created is to create a new commitment variable using $cmt_{\mathsf{new}}()$. In such case, the commitment index is changed, so the old hash is no longer valid.

The `revealCmt` method is translated into two PRISM commands:

- the first command changes the commitment state from committed to revealed and chooses its value at random using the probabilistic updates, similarly as in the `random` expression[22],
- the second command is activated if the commitment is already revealed and in this case its value remains unchanged.

[22]See sec. D11.

The full translation of the `revealCmt` method is described below:

$$\begin{array}{l}
\mathsf{T}(c.\texttt{revealCmt()}) = \\
\{ \\
\quad \sigma_1 = \sigma_{\mathsf{curr}}(); \\
\quad \texttt{mod}.\mathsf{add}( \\
\quad\quad \mathsf{from} : \sigma_{\mathsf{curr}}(), \mathsf{to} : \sigma_{\mathsf{new}}(), \\
\quad\quad \mathsf{guards} : [(\mathsf{allCmt}(c).\mathsf{state} = \mathsf{committed})], \\
\quad\quad \mathsf{updates} : \{1/R : [(\mathsf{allCmt}(c).\mathsf{value}, 0), \\
\quad\quad\quad (\mathsf{allCmt}(c).\mathsf{state}, \mathsf{revealed})], \\
\quad\quad \vdots \\
\quad\quad \mathsf{updates} : 1/R : [(\mathsf{allCmt}(c).\mathsf{value}, R-1), \\
\quad\quad\quad (\mathsf{allCmt}(c).\mathsf{state}, \mathsf{revealed})]\}); \\
\quad \sigma_2 = \sigma_{\mathsf{curr}}(); \\
\quad \texttt{mod}.\mathsf{add}( \\
\quad\quad \mathsf{from} : \sigma_1, \mathsf{to} : \sigma_2, \\
\quad\quad \mathsf{guards} : [(\mathsf{allCmt}(c).\mathsf{state} = \mathsf{revealed})]); \\
\}
\end{array}$$

The translation of `valueOf` and `hashOf` functions is straightforward, since the value of the commitment is stored in the allCmt list and the hash of the commitment is represented by the index of the commitment in this list:

$$\begin{array}{l}
\mathsf{T}(\texttt{valueOf}(c)) = \\
\{ \\
\quad \mathsf{return}\ \mathsf{allCmt}(c).\mathsf{value}; \\
\}
\end{array}$$

$$\begin{array}{l}
\mathsf{T}(\texttt{hashOf}(c)) = \\
\{ \\
\quad \mathsf{return}\ c; \\
\}
\end{array}$$

The verification of the commitment is done by comparing the index of the corresponding commitment variable with the given hash:

$$\begin{array}{l}
\mathsf{T}(\texttt{verCmt}(c,h)) = \\
\{ \\
\quad \mathsf{return}\ (c = h); \\
\}
\end{array}$$

*b) The adversarial usage of commitments:* In addition to the commands described so far, the PRISM model contains also the commands defined to model the adversarial usage of

the commitments. The list of possible adversarial actions with the corresponding PRISM commands are listed below.

Creating a new commitment variable:

$$
\begin{aligned}
&\mathsf{advNewCmt}() = \\
&\{ \\
&\quad \texttt{mod}.\mathsf{add}( \\
&\qquad \mathsf{from} : -1, \mathsf{to} : -1, \\
&\qquad \mathsf{updates} : [(c, \mathit{cmt}_{\mathsf{new}}()), (\mathsf{allCmt}(c).\mathsf{state}, \mathsf{init})]); \\
&\}
\end{aligned}
$$

Creating an already revealed commitment with a particular, chosen value by changing the state of the commitment variable directly from init to revealed and setting the value of the commitment to the chosen value:

$$
\begin{aligned}
&\mathsf{advChosenCmt}(r) = \\
&\{ \\
&\quad \texttt{mod}.\mathsf{add}( \\
&\qquad \mathsf{from} : -1, \mathsf{to} : -1, \\
&\qquad \mathsf{guards} : [(\mathsf{allCmt}(c).\mathsf{state} = \mathsf{init})], \\
&\qquad \mathsf{updates} : [(\mathsf{allCmt}(c).\mathsf{value}, r), \\
&\qquad\quad (\mathsf{allCmt}(c).\mathsf{state}, \mathsf{revealed})]); \\
&\}
\end{aligned}
$$

Creating a random commitment by changing the state of the commitment variable from init to committed:

$$
\begin{aligned}
&\mathsf{advRandomCmt}() = \\
&\{ \\
&\quad \texttt{mod}.\mathsf{add}( \\
&\qquad \mathsf{from} : -1, \mathsf{to} : -1, \\
&\qquad \mathsf{guards} : [(\mathsf{allCmt}(c).\mathsf{state} = \mathsf{init})], \\
&\qquad \mathsf{updates} : [(\mathsf{allCmt}(c).\mathsf{state}, \mathsf{committed})]); \\
&\}
\end{aligned}
$$

Randomly opening the commitment by changing the state of the commitment variable from committed to revealed and choosing the value of the commitment at random:

$$
\begin{aligned}
&\mathsf{advRevealCmt}() = \\
&\{ \\
&\quad \texttt{mod}.\mathsf{add}( \\
&\qquad \mathsf{from} : -1, \mathsf{to} : -1, \\
&\qquad \mathsf{guards} : [(\mathsf{allCmt}(c).\mathsf{state} = \mathsf{committed})], \\
&\qquad \mathsf{updates} : \{1/R : [(\mathsf{allCmt}(c).\mathsf{value}, 0), \\
&\qquad\quad (\mathsf{allCmt}(c).\mathsf{state}, \mathsf{revealed})], \\
&\qquad \vdots \\
&\qquad \mathsf{updates} : 1/R : [(\mathsf{allCmt}(c).\mathsf{value}, R-1), \\
&\qquad\quad (\mathsf{allCmt}(c).\mathsf{state}, \mathsf{revealed})]\}); \\
&\}
\end{aligned}
$$

It is important that the adversary cannot open a commitment from the committed state with a chosen value. It reflects the problem described in sec. VI-D: otherwise, the adversary would be able to arbitrarily change the value of the commitment after the *commit* phase, which is not possible with cryptographic commitments.

*c) Properties of the commitments:* In this section we prove that the PRISM model generated with the described commands indeed implements the cryptographic commitments and preserves their semantics and properties.

**Lemma 9.** *The PRISM model generated by the translation described in sections D12a and D12b preserves the semantics of the ETV symbols related to the commitments (formulas (10)–(14)).*

*Proof.* The translated code for the `randomCmt` method creates a new commitment variable, sets its state to committed and stores its sequential number (index) as $c$. Note that at this stage the value of the commitment is not yet chosen. However this does not break the semantics defined in (10) since at this stage the commitment is in state committed and its value cannot be read.

The value of the commitment can be read only in the revealed state which can be reached only after execution of the `revealCmt` statement. The model generated by $\mathsf{T}(c.\texttt{revealCmt()})$ is presented in fig. 5.

In case the commitment is in the committed state, this command changes its state to revealed and sets its value to one of the numbers $\{0, \ldots, R-1\}$ with equal probabilities. In case the commitment is already in the revealed state, this command does nothing. Therefore the semantic rules (10) and (11) are preserved.

Recall that the value of the commitment is stored under $\mathsf{allCmt}(c).\mathsf{value}$ while $c$ represents its hash. Hence, $\mathsf{T}(\texttt{valueOf}(c))$ and $\mathsf{T}(\texttt{hashOf}(c))$ clearly satisfy (12) and (13).

Lastly, the hash of the commitment is also represented as $c$ which is the sequential number of the corresponding commitment variable. Therefore in order to verify if $h$ is a proper hash of the commitment $c$ it is sufficient to verify if $c = h$. Hence, (14) is also preserved. □

In order to use the models created above as a commitment scheme, we must prove that they provide perfect security with respect to *hiding* and *binding* properties:

- *hiding*: before the *reveal* phase the value of the commitment is *hidden* from the opponent, i.e., no one can learn its value and use this knowledge to make his non-deterministic choices
- *binding*: after the commitment is created, its value cannot be changed

**Lemma 10.** *The commitments defined with the translation commands described above preserve the* hiding *and* binding *properties.*

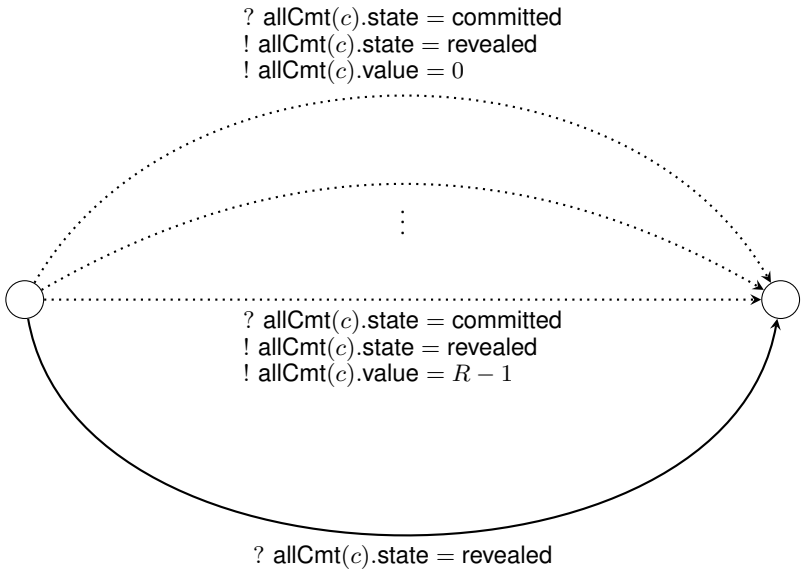



Fig. 5. Translation of the `revealCmt` statement

*Proof.* We first proof the hiding property and then the binding property.

**Hiding.** We expect that once the commitment is created, its value is *hidden* from the opponent, i.e. no one can learn its value before the *revealing* phase. It is especially important in PRISM since every variable in PRISM is public, so once the final value of a variable is set, every PRISM module can *make non-deterministic choices* using the knowledge of this value (see sec. VI-D). In our case the actual random choice is made in the command created by $\mathsf{T}(\texttt{revealCmt})$, so between `randomCmt` and `revealCmt` the final value of the commitment is not yet defined.

Using the same reasoning, if the adversary decides to follow the *honest path* and choose the commitment value at random, then the commitment value is hidden until the `revealCmt` phase from the honest player. Hence the *hiding* property is the commitment scheme defined as above is preserved.

**Binding.** The second property that we expect from the commitment scheme is *binding* which means that once the commitment is created, its value cannot be changed without breaking the correctness of the commitment.

In case of the *honest* commitments commands, the state of a commitment variable can be changed only from init to committed (in the `randomCmt` command) of from committed to revealed (in the `revealCmt` command). So once the commitment is created, its value:

- either is not yet chosen and it only can be set to a random value,
- or is already chosen (the commitment is in the revealed state) and cannot be changed.

The only way to change the commitment value is to create a new commitment using the `randomCmt` command. However, in that case the index of the commitment variable will be changed to a new one, so the commitment will be no longer verifiable with `verCmt`.

Note that very similar reasoning applies to the adversarial commitment commands. Indeed, the only possible actions for the adversary which does not break the commitment are:

- change the state of the commitment from init to committed
- change the state of the commitment from committed to revealed and choose the commitment value at random,
- change the state of the commitment from init directly to revealed and set its value to a particular number.

Hence, once the commitment is created, its value either is already set and cannot be changed, or is not yet chosen and will be determined at random. Also, by creating a new commitment the adversary breaks the old one, since a new commitment variable index is used and it no longer can be verified with `verCmt` with the same hash. Therefore, the commitment scheme described above preserves the *binding* property. □

*13) Compiling the digital signatures:* ETV provides a convenient syntax for digital signatures consisting of the `sign` and `verSig` functions. In MDP, similarly to the commitments, we store the current values of all the signatures in a dedicated list, allSig. The signatures are immutable, i.e., the values stored at a particular index in this list cannot be changed.

Recall that we defined the semantics of the `sign` and `verSig` operations as follows:

$$\frac{}{\begin{array}{l}[[s = \texttt{sign}(x_1, \ldots, x_n)]]_\rho \\ = \rho.S\{(\rho.V(\texttt{sender}), x_1, \ldots, x_n)/s\}\end{array}} \tag{15}$$

$$\frac{\rho.S(s) = (\mathcal{A}, x_1, \ldots, x_n)}{[[\texttt{verSig}(\mathcal{A}, s, x_1, \ldots, x_n)]]_\rho = \texttt{true}} \tag{16}$$

$$\frac{\rho.S(s) \neq (\mathcal{A}, x_1, \ldots, x_n)}{[[\texttt{verSig}(\mathcal{A}, s, x_1, \ldots, x_n)]]_\rho = \texttt{false}} \tag{17}$$

Below we present the two parts of the created model that are related to the signatures. The first part is generated by the translation of `sign` and `verSig` functions from the honest execution of the code, while the second part corresponds to the adversarial usage of the signatures.

*a) The honest usage of the signatures:* We define the translation of the `sign` function as follows:

$$\begin{aligned}
&\mathsf{T}(\texttt{sign}(x_1, \ldots, x_n)) = \\
&\{ \\
&\quad \texttt{mod}.\mathsf{add}( \\
&\quad\quad \mathsf{from} : \sigma_{\mathsf{curr}}(), \mathsf{to} : \sigma_{\mathsf{new}}(), \\
&\quad\quad \mathsf{updates} : [(s, sig_{\mathsf{new}}()), (\mathsf{allSig}(s).\mathsf{user}, \texttt{sender}), \\
&\quad\quad\quad (\mathsf{allSig}(s).\mathsf{value}, (x_1, \ldots, x_n))]); \\
&\}
\end{aligned}$$

In this function we use $sig_{\mathsf{new}}()$ to retrieve a new, unused element from the allSig list, and store its index as $s$. Furthermore, we store the signer and the values being signed at this index in the list. Clearly, this command preserves (15) since it is a direct implementation of it.

The `verSig` function is translated to MDP as follows:

$$\begin{aligned}
&\mathsf{T}(\texttt{verSig}(\mathcal{A}, s, x_1, \ldots, x_n)) = \\
&\{ \\
&\quad \mathsf{return}(\mathsf{allSig}(s).\mathsf{user} = \mathcal{A}) \wedge (\mathsf{allSig}(s).\mathsf{value} = (x_1, \ldots, x_n)); \\
&\}
\end{aligned}$$

In order to verify if $s$ is the correct signature of $\mathcal{A}$ on data $(x_1, \ldots, x_n)$, we compare the values stored in the map: $\mathsf{allSig}(s).\mathsf{user}$ and $\mathsf{allSig}(s).\mathsf{value}$ with $\mathcal{A}$ and $(x_1, \ldots, x_n)$. Again, this command is a direct implementation of (16) and (17), so it preserves the semantics of the `verSig` expression.

*b) The adversarial usage of the signatures:* Analogously to the case of the commitments, we allow the adversary to create new signatures of any data. The command that can be used by the adversary to create a new signature is presented below:

$$\begin{aligned}
&\mathsf{advSign}(x_1, \ldots, x_n) = \\
&\{ \\
&\quad \texttt{mod}.\mathsf{add}( \\
&\quad\quad \mathsf{from} : -1, \mathsf{to} : -1, \\
&\quad\quad \mathsf{updates} : [(s, sig_{\mathsf{new}}()), (\mathsf{allSig}(s).\mathsf{user}, \texttt{sender}), \\
&\quad\quad\quad (\mathsf{allSig}(s).\mathsf{value}, (x_1, \ldots, x_n))]); \\
&\}
\end{aligned}$$

*c) Properties of the signatures:* As it was described in sec. C6, in addition to the formulas (15), (16) and (17) we require from the signatures to satisfy the *unforgeability* property.

**Lemma 11.** *The signature scheme implementation presented in sections D13a and D13b is* unforgeable*, i.e., it is infeasible to produce signatures of other users to data they did not sign.*

*Proof.* Both, honest ($\mathsf{T}(\texttt{sign}(x_1, \ldots, x_n))$) and adversarial (advSign) commands set the user property of the signature to the sender of the command. Hence, it is impossible to produce a signature of the other user. Furthermore, both functions retrieve the new, unused signature variable from the list allSig using $sig_{\mathsf{new}}()$, and it is the only way to modify the signature. Therefore, after the creation, the user and value properties of a given signature cannot be changed. □

*14) The complexity of the translation:* The translation of each expression and statement is linear, so the complexity of the *honest* part of the translation is linear with respect to the length of the ETV program.

On the other hand, the adversarial part of the model contains the *trigger* commands which executes any contract/communication function with any valuation of the arguments. The number of these commands is equal to the number of different valuations of the arguments of the functions. Therefore we can derive the formula for the complexity of the translation as follows:

Let $F_{\mathsf{comm}}$ be the set of all the communication functions, $F_{\mathsf{pay}}$ be the set of all the payable contract functions and $F_{\mathsf{nopay}}$ be the set of all the not payable functions from the contract. Let $f.args$ denote the arguments of function $f$. Furthermore, let $\mathsf{ran}(arg)$ denote the range (the number of possible values) of argument *arg* and let $u_{max}$ will be the maximal value of the contract call. Using this syntax we can provide the following formula for the total number of valuations of function $f$:

$$\mathsf{val}(f) = \begin{cases} \left(\prod_{arg \in f.args} \mathsf{ran}(arg)\right) \cdot (u_{max} + 1) & \text{if } f \in F_{\mathsf{pay}} \\ \prod_{arg \in f.args} \mathsf{ran}(arg) & \text{if } f \in F_{\mathsf{tot}} \end{cases}$$

where $F_{\mathsf{tot}} = (F_{\mathsf{comm}} \cup F_{\mathsf{nopay}})$. Each *trigger* command can be generated in a constant time, so the total time of generation of the trigger commands is proportional to the sum of $\mathsf{val}(f)$ expressions. Hence the total complexity of the translation can be described with the following formula:

$$O\left(len(\mathsf{program}) + \sum_{f \in (F_{\mathsf{comm}} \cup F_{\mathsf{pay}} \cup F_{\mathsf{nopay}})} \mathsf{val}(f)\right)$$

### *E. Case study: Rock-Paper-Scissors*

*1) Introduction:* In this section we present a case study of running *EthVer* on a real life example of the Rock-Paper-Scissors contract from *Delmolino et al., Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab* [17]. In this paper the authors describe the smart contracts created by the attendees of a cryptocurrency course for undergraduate students. The authors document the

typical mistakes student made and suggest the ways to avoid them when designing a secure smart contract.

The Rock-Paper-Scissors is a protocol in which two players choose their inputs from $\{R, P, S\}$ and bet some money (e.g. 1 *finney*) each. The reward of 2 *finney* goes to the winner according to the standard rules of the game or is split between the players in case of a draw. The students created an Ethereum contract code for this protocol and then the code was analyzed for the potential vulnerabilities. The authors pointed out the following bugs in the code:

- if any player sends wrong amount, his money is lost,
- if any player joins the game as the third player, his money is also lost,
- a malicious player can wait until his opponent makes his move and then learn the opponent's choice (since it is a publicly visible variable in the network) and make an according move, which guarantees him a victory.

All of the issues described above are the typical bugs that may arise during designing a smart contract and they can be prevented by careful programming. It turns out that *EthVer* could help the programmer to find and correct each of these bugs in an automated way and in this section we present a step by step analysis of the contract with *EthVer* which leads to the final, correct and secure version of the code.

*2) Original, buggy code of the contract:* Below we present the orginal code of the contract.

```
contract RPS {
  uint(3) num_players;
  address player_address[2];
  uint(3) player_choice[2];
  uint(3) p0;
  uint(3) p1;
  cash(3) reward;

  function player_input(uint(3) choice)
      public payable {
    if (num_players < 2 && msg.value == finney(1)) {
      reward = reward + msg.value;
      player_address[num_players] = msg.sender;
      player_choice[num_players] = choice;
      num_players = num_players + 1;
    }
  }

  function finalize() public {
    p0 = player_choice[0];
    p1 = player_choice[1];
    if (((3 + p0 - p1) % 3) == 1) {
      player_address[0].transfer(reward);
    }
    if (((3 + p0 - p1) % 3) == 2) {
      player_address[1].transfer(reward);
    }
    if (((3 + p0 - p1) % 3) == 0) {
      player_address[0].transfer(reward/2);
      player_address[1].transfer(reward/2);
    }
  }

}

scenario A {
  bool endA;
  uint(3) choiceA;
  choiceA = random(3);
  player_input.sendTransaction(choiceA, {value: 1});
  wait(num_players == 2, 1);
  finalize.sendTransaction({value: 0});
  endA = true;
}

scenario B {
  bool endB;
  uint(3) choiceB;
  choiceB = random(3);
  player_input.sendTransaction(choiceB, {value: 1});
  wait(num_players == 2, 1);
  finalize.sendTransaction({value: 0});
  endB = true;
}
```

Listing 6. RPS v1

Listing 6 depicts the original, buggy code of the Rock-Paper-Scissors contract[23]. It is written in the ETV language which is is broadly described in chapter III. This code can be passed as the input file to *EthVer* to generate the Ethereum contract code and a PRISM model for it, which in turn can be used to verify the PRISM properties.

In order to create the PRISM properties to verify, we must first define, what real properties we expect from the model. In the real Rock-Paper-Scissors game an honest player has 1/3 probability of winning, 1/3 probability of losing and 1/3 probability of a draw. The protocol is modeled in PRISM as a Markov Decision Process (MDP). Recall that for MDPs, PRISM allows us to verify the *minimum probability* and the *maximum probability* properties, where the probability is taken over all random choices of the model and the minimum/-maximum is taken over all the non-deterministic choices of the model. Therefore, we can construct the following PRISM properties to verify[24]:

```
Pmin=? [ F endA & balance0 = 2 ]
Pmax=? [ F endA & balance0 = 2 ]
Pmin=? [ F endA & balance0 = 1 ]
Pmax=? [ F endA & balance0 = 1 ]
Pmin=? [ F endA & balance0 = 0 ]
Pmax=? [ F endA & balance0 = 0 ]
```

All the properties above should evaluate to 1/3 and can be read as: *What is the minimal/maximal probability that player 0 wins/draws/loses?* After we run the verification of these properties on the generated model in PRISM, it turns out that the properties indeed evaluate to 1/3 each. This means that in case of a game between two honest players, each of them has equal chances of winning, losing and a draw.

In order to fully verify the correctness and security of a contract, we must consider a game between an honest player and a malicious player and make sure that the malicious player cannot increase his chances of winning or a draw. This can be simplified to two properties for a game between an honest player 0 and a malicious player 1:

- the probability of player 0 win is at least equal to 1/3,

[23]Some technical details of the code (like the constants declarations) are omitted for readability. For the full version of all the code examples, please refer to the official repository of *EthVer* at github.com/lukmaz/ethver.

[24]Here and later we omit the symmetric properties for player 1.

- the probability of player 0 win or a draw is at least equal to 2/3.

These properties can be written as PRISM properties and then verified in the honest (`ADVERSARY = -1`) and the adversarial (`ADVERSARY = 1`) modes[25]. The results of the verification are shown in the table II.

| property | target | result (honest mode) | result (adversarial mode) |
|---|---|---|---|
| `Pmin=? [ F endA & balance0 >= 2 ]` | 1/3 | 1/3 | 0 |
| `Pmin=? [ F endA & balance0 >= 1 ]` | 2/3 | 2/3 | 0 |

TABLE II
PROPERTIES FOR RPS V1

The obtained minimal probabilities equal to 0 means that in the pessimistic scenario the adversary can decrease the chances of winning or a draw of the honest player to 0.

*3) Early interruption of the protocol:* We have seen that the contract code presented in the previous section is secure in the honest mode, however it contains some bugs which make it vulnerable to attacks by the malicious player. In order to find the bugs, we can start debugging RPS v1 with verifying if the final distribution of money corresponds to the chosen values of `p0`, `p1`. To find the problematic scenario, we will use the powerful feature of PRISM for generating a witness/counterexample for the *reachability* properties. In order to use it, we must formulate a reachability property describing the unwanted state, e.g., *eventually, the player 0 finishes with no money while the chosen values were the same*:

```
E [ F endA & balance0 = 0 & p0 = p1 ]   (18)
```

This should never happen because `p0 = p1` means that it is a draw, so player 0 should end up with `balance0 = 1` and the property should evaluate to false.

If we verify the property in PRISM in the *honest mode* (`ADVERSARY = -1`), it indeed evaluates to false. However if we do the same in the *adversarial mode* (`ADVERSARY = 1`), then the property evaluates to true and PRISM generates a *witness* — the path to reach this undesirable state. Figure 6 shows the witness path. A quick look on the functions being executed in the first 20 steps gives an immediate answer, what is wrong[26]. The order of the contract functions execution is as follows:

- player 1 call of `player_input`
- player 1 call of `finalize`
- ...

[25]Recall that (`ADVERSARY = -1`) stands for the honest execution, (`ADVERSARY = 0`) is for malicious player 0 vs honest player 1 and (`ADVERSARY = 1`) is for honest player 0 vs malicious player 1.

[26]In the *Action* column the actions with names ending with a number 0 or 1 correspond to the moments when players send the appropriate transactions to the contract. The actions without a trailing number in the name correspond to the actual execution of the function calls by the contract.

| Step | | | | | |
|---|---|---|---|---|---|
| Action | # | time_el... | choice | contr_sen... | fin |
| | 0 | 0 | 0 | 0 | |
| player1 | 1 | 0 | 0 | 0 | |
| player0 | 2 | 0 | 0 | 0 | |
| player0 | 3 | 0 | 0 | 0 | |
| player0 | 4 | 0 | 0 | 0 | |
| [broadcast_player_input1] | 5 | 0 | 0 | 0 | |
| [broadcast_player_input0] | 6 | 0 | 0 | 0 | |
| [broadcast_finalize1] | 7 | 0 | 0 | 0 | |
| [broadcast_player_input] | 8 | 0 | 1 | 1 | |
| contract | 9 | 0 | 1 | 1 | |
| contract | 10 | 0 | 1 | 1 | |
| contract | 11 | 0 | 1 | 1 | |
| contract | 12 | 0 | 1 | 1 | |
| contract | 13 | 0 | 1 | 1 | |
| contract | 14 | 0 | 1 | 1 | |
| contract | 15 | 0 | 1 | 1 | |
| contract | 16 | 0 | 1 | 1 | |
| contract | 17 | 0 | 1 | 1 | |
| [broadcast_finalize] | 18 | 0 | 1 | 1 | |
| contract | 19 | 0 | 1 | 1 | |
| contract | 20 | 0 | 1 | 1 | |
| contract | 21 | 0 | 1 | 1 | |

Fig. 6. The witness for the property (18)

Function `finalize` (broadcast by player 1) is executed before the execution of player 0 `player_input` function call. This is an obvious drawback of the contract — calling `finalize` function should not be allowed before both `player_input` functions have been executed.

*a) Fix 1:* The above flaw in the contract can be easily fixed by adding an additional condition at the beginning of the `finalize` function:

```
function finalize() public {
  if (num_players == 2) {
    ...
```

After this fix we can rerun the verification. It turns out that the reachability property (18) is still satisfied, which means that the final distribution of money still does not correspond to the chosen values of `p0` and `p1`. If we take a look at the witness generated by PRISM for this property, we will notice that this time the critical path is as follows:

- player 0 call of `player_input`
- time step
- player 1 call of `finalize`
- ...

The alarming thing is that the `time_step` occurs before player 1 call of `player_input`. It means that player 1 does not execute his `player_input` step within the given time period. This is another obvious flaw in the contract — player 0 should not lose his money just because his opponent didn't perform the required action on time.

*b) Fix 2:* The shortcoming mentioned above can be fixed by adding a *refund* possibility for the first player, if the second player didn't join the game on time:

```
function finalize() public {
  if (time_elapsed > 0 && num_players == 1)
    player_address[0].transfer(reward);
  ...
```

After rerunning the verification we can see that this fix also does not solve the problem completely — the reachability property (18) is still satisfied. Now the verification generates

| Step | | | | | |
|---|---|---|---|---|---|
| Action | # | time_el... | choice | contr_sender | fi |
| | 0 | 0 | 0 | 0 | |
| player1 | 1 | 0 | 0 | 0 | |
| player0 | 2 | 0 | 0 | 0 | |
| player0 | 3 | 0 | 0 | 0 | |
| player0 | 4 | 0 | 0 | 0 | |
| [broadcast_player_input1] | 5 | 0 | 0 | 0 | |
| [broadcast_player_input0] | 6 | 0 | 0 | 0 | |
| [broadcast_player_input] | 7 | 0 | 1 | 1 | |
| contract | 8 | 0 | 1 | 1 | |
| contract | 9 | 0 | 1 | 1 | |
| contract | 10 | 0 | 1 | 1 | |
| contract | 11 | 0 | 1 | 1 | |
| contract | 12 | 0 | 1 | 1 | |
| contract | 13 | 0 | 1 | 1 | |
| contract | 14 | 0 | 1 | 1 | |
| contract | 15 | 0 | 1 | 1 | |
| contract | 16 | 0 | 1 | 1 | |
| [broadcast_player_input1] | 17 | 0 | 1 | 1 | |
| [broadcast_player_input] | 18 | 0 | 1 | 1 | |
| contract | 19 | 0 | 1 | 1 | |
| contract | 20 | 0 | 1 | 1 | |

Fig. 7. The witness for the property (18) run on RPS v1 model after Fix 1 and Fix 2

another critical path (see fig. 7):

- player 1 call of `player_input`
- player 1 call of `player_input`
- ...

This emphasizes another drawback of the protocol: a malicious player 1 can execute the `player_input` function twice before player 0 joins the game. This scenario is analogous to the situation in which two players joins the game and then player 0 tries to join the game as a **third** one. This cannot be prevented, since even if the cautious player checks the number of joined players before his move, there still is a possibility that two players calls the `player_input` function simultaneously. In such case the decision of choosing the first function call to process depends on several random factors and the player who joins last will lose his bet while staying outside of the game.

*c) Fix 3:* To mitigate the last problem, we can add another *refund* mechanism to the `player_input` function which sends the bet back to the transaction sender, if the game is already full:

```
function player_input() public {
  if (num_players >= 2)
    msg.sender.transfer(msg.value);
  ...
```

The complete contract code after fixes 1, 2, 3 is depicted in the listing 7 and will be referred to as RPS v2[27].

```
contract RPS {
  uint(3) num_players;
  address player_address[2];
  uint(3) player_choice[2];
  uint(3) p0;
  uint(3) p1;
  cash(3) reward;

  function player_input(uint(3) choice)
        public payable {
    if (num_players >= 2 || msg.value != finney(1)) {
      msg.sender.transfer(msg.value);
    }
    if (num_players < 2 && msg.value == finney(1)) {
      reward = reward + msg.value;
      player_address[num_players] = msg.sender;
      player_choice[num_players] = choice;
      num_players = num_players + 1;
    }
  }

  function finalize() public {
    if (time_elapsed > 0 && num_players == 1) {
      player_address[0].transfer(reward);
    }
    if (num_players == 2) {
      p0 = player_choice[0];
      p1 = player_choice[1];
      if (((3 + p0 - p1) % 3) == 1) {
        player_address[0].transfer(reward);
      }
      if (((3 + p0 - p1) % 3) == 2) {
        player_address[1].transfer(reward);
      }
      if (((3 + p0 - p1) % 3) == 0) {
        player_address[0].transfer(reward/2);
        player_address[1].transfer(reward/2);
      }
    }
  }
}

scenario A {
  bool endA;
  uint(3) choiceA;
  choiceA = random(3);
  player_input.sendTransaction(choiceA, {value: 1});
  wait(num_players == 2, 1);
  finalize.sendTransaction({value: 0});
  endA = true;
}

scenario B {
  bool endB;
  uint(3) choiceB;
  choiceB = random(3);
  player_input.sendTransaction(choiceB, {value: 1});
  wait(num_players == 2, 1);
  finalize.sendTransaction({value: 0});
  endB = true;
}
```

Listing 7. RPS v2

[27]The full code of the intermediate versions of the contract after Fix 1 and Fix 2 can be found in the project repository as `rpsv1a.etv` and `rpsv1b.etv`.

*4) Cryptographic commitments:* After fixes 1, 2 and 3 the reachability property (18) is no longer satisfied. This means that the problem of lack of connection between the chosen values and the final money distribution has disappeared. However, the properties from table II still do not evaluate to the desired probabilities.

This time, the problem is more subtle. In order to address it, we must take a closer look into how the transactions are being processed in a blockchain-based system. When a user calls a contract function, he creates a transaction, broadcasts it to a peer-to-peer network and eventually the transaction gets published in one of the upcoming blocks. This in particular means that all the data contained in the transactions (like, e.g., the arguments of the function calls) are publicly visible to all the network nodes and cannot be changed. Thus, a malicious player in our Rock-Paper-Scissors game can wait until his opponent posts his move and then react with the appropriate move, which guarantees him a victory.

This feature of the blockchain-based system is reflected in a PRISM model checker in the following way. Every variable in PRISM is *public*, which means that once the value of a variable is set, it can influence the non-deterministic choices of the next model steps[28]. In other words, after an honest player makes his move, its result is publicly known and the next, non-deterministic moves of the malicious player can take advantage of this knowledge. That is the reason why the properties from the table II evaluate to 0 — in these properties we compute the *minimal probability*, and the minimum is taken over all the non-deterministic choices in the model.

The first line of defense against such pitfalls are the cryptographic commitments. In order to prevent the chosen value from being visible to others, the value is chosen in secret and only the commitment is published. In ETV language, we provide the `cmt_uint` type for commitments as well as functions `randomCmt` and `verCmt` which implement this functionality (cf. chapter D). Thanks to this mechanism, the exact value is not chosen at the moment of creating the commitment, but it is postponed until the *open* phase — the moment when the value of the variable is read.

The Rock-Paper-Scissors code updated with the commitments is presented in the listing 8.

```
contract RPS {
  uint(3) num_players;
  address player_address[2];
  uint(3) player_choice[2];
  uint(3) p0;
  uint(3) p1;
  cash(3) reward;
  uint(2) player_num;
  hash commitment[2];

  bool committed[2];
  bool has_revealed[2];

  function player_input(hash h) public payable {
    if (num_players >= 2 || msg.value != finney(1)) {
      msg.sender.transfer(msg.value);
    }
    else if (num_players < 2
          && msg.value == finney(1)) {
      reward = reward + msg.value;
      player_address[num_players] = msg.sender;
      commitment[num_players] = h;
      committed[num_players] = true;
      num_players = num_players + 1;
    }
  }

  function open(cmt_uint(3) cmt) public {
    if (msg.sender == player_address[0]) {
      player_num = 0;
    }
    else if (msg.sender == player_address[1]) {
      player_num = 1;
    }
    if (committed[player_num]
        && !has_revealed[player_num]
        && verCmt(cmt, commitment[player_num])) {
      has_revealed[player_num] = true;
      player_choice[player_num] = valueOf(cmt);
    }
  }

  function finalize() public {
    if (time_elapsed > 0 && num_players == 1) {
      player_address[0].transfer(reward);
    }
    if (has_revealed[0] && has_revealed[1]) {
      p0 = player_choice[0];
      p1 = player_choice[1];
      if ((((3 + p0) - p1) % 3) == 1) {
        player_address[0].transfer(reward);
      }
      if ((((3 + p0) - p1) % 3) == 2) {
        player_address[1].transfer(reward);
      }
      if ((((3 + p0) - p1) % 3) == 0) {
        player_address[0].transfer(reward/2);
        player_address[1].transfer(reward/2);
      }
    }
  }
}

scenario A {
  bool endA;
  cmt_uint(3) cmtA;
  cmtA.randomCmt();

  player_input.sendTransaction(hashOf(cmtA),
      {value: 1});
  wait(num_players == 2, 1);
  if (num_players == 2) {
    cmtA.revealCmt();
    open.sendTransaction(cmtA, {value: 0});
    wait(has_revealed[0] && has_revealed[1], 2);
  }
  finalize.sendTransaction({value: 0});
  endA = true;
}

scenario B {
  bool endB;
  cmt_uint(3) cmtB;
  cmtB.randomCmt();

  player_input.sendTransaction(hashOf(cmtB),
      {value: 1});
  wait(num_players == 2, 1);
  if (num_players == 2) {
    cmtB.revealCmt();
```

[28]The problem of public variables in PRISM has already been discussed in sec. VI-D.

```
    open.sendTransaction(cmtB, {value: 0});
    wait(has_revealed[0] && has_revealed[1], 2);
  }
  finalize.sendTransaction({value: 0});
  endB = true;
}
```

Listing 8. RPS v3

*5) Misaligned incentives:* Even with the cryptographic commitments implemented, the probabilities from the table II still evaluate to 0. To find a bug we will once again use the reachability property (18) to verify if the final distribution of money corresponds to the chosen values of `p0` and `p1`. It turns out that the property (18) evaluates to true and PRISM provides us another witness for it. The critical path this time is:

- player 0 call of `player_input`
- time step
- player 1 call of `player_input`
- player 0 call of `finalize`
- ...

In this scenario the player 1 does not join the game within a given time window. We have already handled this situation in Fix 2, however that fix no longer works, because now the protocol has 2 phases: *commit* and *open*.

If we look carefully at the interleaving above, we can see that the problem is that the malicious player 1 can reject to call `player_input` function on time, wait until player 0 starts the *refund procedure* and then call `player_input` which potentially can be executed by the network before `finalize`. In such case the refund procedure from `finalize` function fails, because at that time `num_players` is already equal to 2.

*a) Fix 4:* One possible remedy for the problem mentioned above is to change the refund condition to:

```
function finalize() public {
  if (time_elapsed > 0 && committed0 && !committed1)
    player_address[0].transfer(reward);
  ...
```

and prohibit players from committing after the time step:

```
function player_input() public payable {
  if (num_players >= 2 || msg.value != 1
        || time_elapsed > 0)
    msg.sender.transfer(msg.value);
  ...
```

This solves the problem described in the last paragraph, however after this fix the reachability property (18) is still satisfied, which means that another unwanted interleaving is possible. This time the critical path looks as as follows:

- player 0 call of `player_input`
- player 1 call of `player_input`
- player 0 call of `open`
- time step
- time step
- player 0 call of `finalize`
- ...

In this case both players join the game by executing the `player_input` function and then only player 0 opens his commitment on time. This is a realistic scenario when a malicious player 1 waits until player 0 opens his commitment and then (knowing already the result of the game) refuses to open his commitment.

*b) Fix 5:* To prevent from this behavior, we can introduce another refund mechanism, which transfers the whole reward to one player in case his opponent refuses to open his commitment within a given time frame (two time steps):

```
function finalize() public {
  if (time_elapsed > 1 && has_revealed0
        && !has_revealed1) {
    player_address[0].transfer(reward);
  ...
```

The final version of the contract after all the fixes is presented in the listing 9.

```
contract RPS {
  uint(3) num_players;
  address player_address[2];
  uint(3) player_choice[2];
  uint(3) p0;
  uint(3) p1;
  cash(3) reward;
  uint(2) player_num;
  hash commitment[2];

  bool committed[2];
  bool has_revealed[2];

  function player_input(hash h) public payable {
    if (num_players >= 2 || msg.value != finney(1)
        || time_elapsed > 0) {
      msg.sender.transfer(msg.value);
    }
    else if (num_players < 2
        && msg.value == finney(1)
        && time_elapsed == 0) {
      reward = reward + msg.value;
      player_address[num_players] = msg.sender;
      commitment[num_players] = h;
      committed[num_players] = true;
      num_players = num_players + 1;
    }
  }

  function open(cmt_uint(3) cmt) public {
    if (msg.sender == player_address[0]) {
      player_num = 0;
    }
    else if (msg.sender == player_address[1]) {
      player_num = 1;
    }
    if (committed[player_num]
        && !has_revealed[player_num]
        && verCmt(cmt, commitment[player_num])) {
      has_revealed[player_num] = true;
      player_choice[player_num] = valueOf(cmt);
    }
  }


  function finalize() public {
    if (time_elapsed > 1 && has_revealed[0]
        && !has_revealed[1]) {
      player_address[0].transfer(reward);
    }
    else if (time_elapsed > 1 && !has_revealed[0]
```

```
        && has_revealed[1]) {
      player_address[1].transfer(reward);
    }
    else if (time_elapsed > 0 && committed[0]
        && !committed[1]) {
      player_address[0].transfer(reward);
    }
    else if (time_elapsed > 0 && !committed[0]
        && committed[1]) {
      player_address[1].transfer(reward);
    }

    else if (has_revealed[0] && has_revealed[1]) {
      p0 = player_choice[0];
      p1 = player_choice[1];
      if ((((3 + p0) - p1) % 3) == 1) {
        player_address[0].transfer(reward);
      }
      if ((((3 + p0) - p1) % 3) == 2) {
        player_address[1].transfer(reward);
      }
      if ((((3 + p0) - p1) % 3) == 0) {
        player_address[0].transfer(reward/2);
        player_address[1].transfer(reward/2);
      }
    }
  }
}

scenario A {
  bool endA;
  bool joinedA;
  cmt_uint(3) cmtA;
  cmtA.randomCmt();

  player_input.sendTransaction(hashOf(cmtA),
      {value: 1});
  wait(num_players == 2, 1);
  if (num_players == 2 && (player_address[0] == this
      || player_address[1] == this)) {
    joinedA = true;
    cmtA.revealCmt();
    open.sendTransaction(cmtA, {value: 0});
    wait(has_revealed[0] && has_revealed[1], 2);
  }
  finalize.sendTransaction({value: 0});
  endA = true;
}

scenario B {
  bool endB;
  cmt_uint(3) cmtB;
  cmtB.randomCmt();

  player_input.sendTransaction(hashOf(cmtB),
      {value: 1});
  wait(num_players == 2, 1);
  if (num_players == 2 && (player_address[0] == this
      || player_address[1] == this)) {
    cmtB.revealCmt();
    open.sendTransaction(cmtB, {value: 0});
    wait(has_revealed[0] && has_revealed[1], 2);
  }
  finalize.sendTransaction({value: 0});
  endB = true;
}
```

Listing 9. RPS v4

*6) Final properties to verify:* Let us go back to our initial properties and verify them with the code after all the fixes:

| property | target | result (honest mode) | result (adversarial mode) |
|---|---|---|---|
| `Pmin=? [ F endA & balance0 >= 2 ]` | 1/3 | 1/3 | **0** |
| `Pmin=? [ F endA & balance0 >= 1 ]` | 2/3 | 2/3 | 2/3 |

We can see that the second property evaluates to the desired probability of 2/3, but the first property evaluates to 0. Recall that the first property can be read as *player 0 wins with probability at least 1/3*. The value of 0 follows from the fact, that player 1 can interrupt the protocol before sending any money to the contract. In such case, the *refund* mechanism pays back to player 0 only his own bet of value 1 and this is fine. What we really want to verify is what is the minimal probability of winning of player 0 *assuming that both players have successfully joined the game*. This can be expressed with a following PRISM property:

```
filter(min, Pmin=? [ F balance0 >= 2 ], joinedA
    & !cmtA_revealed & !cmtB_revealed)
```

which can be read as *what is the minimal probability of winning of player 0 assuming that player 0 is in the game, both players have entered the game and none of the players have revealed their choice yet*.[29] This property indeed evaluates to 1/3, so after all the fixes we finally reached the version of the contract which satisfies our initial conditions. The final set of properties and the results of verification are shown in the table below:

| property | target | result (honest mode) | result (adversarial mode) |
|---|---|---|---|
| `filter(min, Pmin=? [ F balance0 >= 2 ], joinedA & !cmtA_revealed & !cmtB_revealed)` | 1/3 | 1/3 | 1/3 |
| `Pmin=? [ F endA & balance0 >= 1 ]` | 2/3 | 2/3 | 2/3 |

*7) Summary:* The authors of [17] analyze the buggy Rock-Paper-Scissors contract *by hand*. They address multiple design bugs in the contract, describe how to fix them and formulate a set of *best programming practices* which should be applied to avoid similar bugs in smart contracts.

In our experiment we verified the original contract with *EthVer* and the verifier found each of the bugs described by the authors in a fully automated way, without leveraging the *best programming practices*. Moreover, it provided the *witness* path for each found bug which made it much easier for the programmer to debug the contract. In other words,

[29]The `filter(op, prop, states)` syntax of PRISM is used to verify the property `prop` for all the paths starting from the set of initial states `states` and return the result aggregated using `op` operator over all the paths. The `joinedA`, `!cmtA_revealed` and `!cmtB_revealed` conditions for initial states are necessary in order to ensure that in the initial states the random choices are not yet done. Otherwise the property will include the paths starting from the states after the choices are already done and player 1 wins.

if the students had used *EthVer* to verify their contract, they would have avoided all the bugs contained in the original version of the contract, despite that they did not know the *best programming practices*.

### F. Case study: Micropayments

*1) Introduction:* In this section we present the Micropay 1 protocol from *Rafael Pass, abhi shelat, Micropayments for Decentralized Currencies* [30]. The protocol was originally published at the *22nd ACM SIGSAC Conference on Computer and Communications Security* and it was vulnerable to so called *front-running attack*. The bug was discovered by us and independently by Joseph Bonneau. After being notified about the bug, the authors published the fixed version of the protocol which mitigates this attack [31].

*2) Original contract code:* Below we present the original code of the contract.

```
contract Micropay {
  cash(2) pot = finney(1);
  uint(2) r1_loc;
  address escrowAddress;

  constructor() {
    escrowAddress = msg.sender;
  }

  function release(cmt_uint(2) rel_c_cmt,
      uint(2) rel_r2, hash rel_c_hash,
      signature(hash, uint(2), address) rel_sigma)
      public {
    r1_loc = valueOf(rel_c_cmt);
    if (verCmt(rel_c_cmt, rel_c_hash) &&
        (r1_loc == rel_r2) &&
        verSig(escrowAddress, rel_sigma,
          (rel_c_hash, rel_r2, msg.sender))) {
      if (pot > 0) {
        msg.sender.transfer(finney(1));
        pot = pot - finney(1);
      }
    }
  }
}

communication {
  bool payment_requested;
  bool payment_issued;

  uint(2) comm_r2;
  hash comm_c_hash;
  address comm_a;
  signature(hash, uint(2), address) comm_sigma;

  function payment_request(hash req_c_hash) public {
    comm_c_hash = req_c_hash;
    comm_a = msg.sender;
    payment_requested = true;
  }

  function payment_issuance(hash iss_c_hash,
      uint(2) iss_r2, address iss_a) public {
    comm_r2 = iss_r2;
    comm_sigma = sign(iss_c_hash, iss_r2, iss_a);
    payment_issued = true;
  }

}

scenario U {
  uint(2) r2;
  bool endU;

  wait(payment_requested, 1);
  if (payment_requested) {
    r2 = random(2);
    payment_issuance.sendCommunication(
        comm_c_hash, r2, comm_a);
  }

  endU = true;
}

scenario M {
  cmt_uint(2) r1_cmt;
  uint(2) r1_val;
  uint(2) r2_loc;
  signature(hash, uint(2), address) sigma_loc;
  bool winM;
  bool endM;
  bool signature_verified;

  address merchantAddress = this;

  r1_cmt.randomCmt();

  payment_request.sendCommunication(hashOf(r1_cmt));
  wait(payment_issued, 1);
  if (payment_issued && time_elapsed < 1) {
    r2_loc = comm_r2;
    sigma_loc = comm_sigma;
    if (verSig(escrowAddress, sigma_loc,
        (hashOf(r1_cmt), r2_loc, merchantAddress))) {
      signature_verified = true;
      r1_cmt.revealCmt();
      r1_val = valueOf(r1_cmt);
      if (r1_val == r2_loc) {
        winM = true;
        release.sendTransaction(r1_cmt, r2_loc,
            hashOf(r1_cmt), sigma_loc, {value: 0});
      }
    }
  }
  endM = true;
}
```

Listing 10. The contract code for the original Micropay 1 protocol

For this protocol we want to verify the following property: *whenever a user U issues a payment, the merchant M gets paid with probability* $1/N$. This translates to the PRISM properties[30]:

```
Pmin=? [ F endM & balance1 >= 1 ]
Pmax=? [ F endM & balance1 >= 1 ]
```

These two properties correspond to the *honest mode* — if both players follow the protocol, then either of the probabilities should be equal to $1/N$. However, besides the *honest mode* verification, we would like to verify if

- if merchant M is malicious, then he cannot increase his chances of getting money.
- if user U is malicious, then he cannot decrease the probability of getting money by merchant M,

[30]Recall that for non-deterministic models in PRISM we must use `Pmin`/`Pmax` operator which computes the minimum/maximum probability over all the non-deterministic choices of the model.

This leads to the total of 4 PRISM properties listed in the table below (to reduce the computational complexity, we perform all the computations for $N = 2$):

| property | adversary | target | result |
|---|---|---|---|
| `Pmin=? [ F endM & balance1 >= 1 ]` | None | 1/2 | 1/2 |
| `Pmax=? [ F endM & balance1 >= 1 ]` | None | 1/2 | 1/2 |
| `Pmax=? [ F balance1 >= 1 ]` | M | 1/2 | 1/2 |
| `Pmax=? [ F endM & signature_verified & balance1 < 1 ]` | U | 1/2 | **1** |

Note that in the last property, we have negated the condition: instead of computing *the minimal probability that the signature of U is correct and M gets paid*, we are computing *the maximal probability, that the signature of U is correct and M does not get paid*. This is only because the second property is easier expressible in the PRISM properties language.

The last column of the table presents the results of the verification in PRISM of the model generated by *EthVer* from the initial contract code. It turns out that the first three properties evaluate to the correct value of $1/2$, while the last one evaluates to 1. In order to investigate the problem, we will use the method known from the previous chapter for generating a witness for unwanted path in PRISM. We will use the following reachability property:

```
E [ F endM & balance1 < 1 & winM ]
```

This property can be read as: *merchant M finishes with* `balance1 = 0` *despite that he has revealed his winning ticket*.

After verifying this property in PRISM, it turns out that the property is true and moreover PRISM provides the following interleaving for the witness:

- merchant M execution of `payment_request`
- user U execution of `payment_issuance`
- user U call of `release` contract function
- ...

This illustrates a real flaw in the protocol: a front-running attack. In this attack a malicious user U successfully calls a `release` function of the contract and receives the reward designed for M. This is possible because U can produce a winning ticket by simulating the merchant protocol and generating the merchant-role random value of his choice (equal to his own user-role random value) and the corresponding commitment. He can then broadcast his version of the `release` transaction which can be processed before the `release` transaction created by merchant M. Moreover, user U can monitor the network and wait until M's transaction `release` appears in the network and only then react with broadcasting his own transaction. In such case there are two concurrent transactions in the network and among them only one can be confirmed. Which transaction will be chosen depends on several random factors. In fact, if U has good enough network setup, he can force his transaction to be accepted before the M's one with high probability.

*3) Mitigating the front-running attack:* The front-running attack can be mitigated using a penalty deposit. At the beginning of the protocol the user U deposits some amount of money (larger than the amount of payment). This deposit can be released in one of the two following ways:

- after some time $\Delta t$ the deposit goes back to U,
- the deposit is *burned* if one provides two competitive winning tickets.

In case user U tries to perform a front-running attack, the merchant M will have the knowledge of two competitive winning tickets and he could use the *burn* function to penalize U. If U honestly follows the protocol, he can get the deposit back after the appropriate time period has passed. The complete code of the Micropay contract updated with this deposit penalty mechanism is depicted in listing 11.

```
contract Micropay {
  cash(2) pot = finney(1);
  uint(2) r1_loc;
  address escrowAddress;

  uint(3) winning_tickets = 0;
  cash(3) deposit = finney(2);
  hash winning_hash;

  constructor() {
    escrowAddress = msg.sender;
  }

  function release(cmt_uint(2) rel_c_cmt,
      uint(2) rel_r2, hash rel_c_hash,
      signature(hash, uint(2), address) rel_sigma)
      public {
    r1_loc = valueOf(rel_c_cmt);
    if (verCmt(rel_c_cmt, rel_c_hash) &&
        (r1_loc == rel_r2) &&
        verSig(escrowAddress, rel_sigma,
            (rel_c_hash, rel_r2, msg.sender))) {
      if (winning_tickets == 0) {
        winning_tickets = 1;
        winning_hash = rel_c_hash;
      }
      if (winning_tickets == 1
          && rel_c_hash != winning_hash)
      {
        winning_tickets = 2;
      }
      if (pot > 0) {
        msg.sender.transfer(finney(1));
        pot = pot - finney(1);
      }
    }
  }

  function burn() public {
    if ((winning_tickets >= 2)
        && (deposit >= finney(2))) {
      "null".transfer(finney(2));
      deposit = deposit - finney(2);
    }
  }

  function release_deposit() public {
    if (time_elapsed >= 2
        && deposit >= finney(2)) {
      escrowAddress.transfer(finney(2));
      deposit = deposit - finney(2);
```

```
    }
  }
}

communication {
  bool payment_requested;
  bool payment_issued;

  uint(2) comm_r2;
  hash comm_c_hash;
  address comm_a;
  signature(hash, uint(2), address) comm_sigma;

  function payment_request(hash req_c_hash) public {
    comm_c_hash = req_c_hash;
    comm_a = msg.sender;
    payment_requested = true;
  }

  function payment_issuance(hash iss_c_hash,
      uint(2) iss_r2, address iss_a) public {
    comm_r2 = iss_r2;
    comm_sigma = sign(iss_c_hash, iss_r2, iss_a);
    payment_issued = true;
  }

}

scenario U {
  uint(2) r2;
  bool endU;

  wait(payment_requested, 1);
  if (payment_requested) {
    r2 = random(2);
    payment_issuance.sendCommunication(
        comm_c_hash, r2, comm_a);
  }
  wait(false, 2);
  release_deposit.sendTransaction({value: 0});

  endU = true;
}

scenario M {
  cmt_uint(2) r1_cmt;
  uint(2) r1_val;
  uint(2) r2_loc;
  signature(hash, uint(2), address) sigma_loc;
  bool winM;
  bool endM;
  bool signature_verified;

  address merchantAddress = this;

  r1_cmt.randomCmt();

  payment_request.sendCommunication(hashOf(r1_cmt));
  wait(payment_issued, 1);
  if (payment_issued && time_elapsed < 1) {
    r2_loc = comm_r2;
    sigma_loc = comm_sigma;
    if (verSig(escrowAddress, sigma_loc,
        (hashOf(r1_cmt), r2_loc, merchantAddress))) {
      signature_verified = true;
      r1_cmt.revealCmt();
      r1_val = valueOf(r1_cmt);
      if (r1_val == r2_loc) {
        winM = true;
        release.sendTransaction(r1_cmt, r2_loc,
            hashOf(r1_cmt), sigma_loc, {value: 0});
        wait(balance1 >= 1, 1);
        if ((balance1 < 1)
            && (winning_tickets >= 2)) {
          burn.sendTransaction({value: 0});
        }
      }
    }
  }
  endM = true;
}
```

Listing 11. The contract code of the fixed v2 version of the Micropay 1 protocol

In order to check if this contract is resilient to the front-running attack, we must slightly modify the properties: now in case of the honest execution of the protocol the user U finishes with `balance0 = 2` (we set the deposit to 2). In case of a successful attack, the merchant M does not receive his payment, however then U will end up with `balance0 = 0`. Therefore we must verify if a malicious user U cannot increase the chances that *merchant M does not get paid while his ticket is correctly signed by U and the user U remains not punished by losing his deposit*. Thus, the final version of properties for this protocol are as follows:

| property | adversary | target | result |
|---|---|---|---|
| `Pmin=? [ F endM & balance1 >= 1 ]` | None | 1/2 | 1/2 |
| `Pmax=? [ F endM & balance1 >= 1 ]` | None | 1/2 | 1/2 |
| `Pmax=? [ F balance1 >= 1 ]` | M | 1/2 | 1/2 |
| `Pmax=? [ F endM & signature_verified & balance0 >= 2 & balance1 < 1 ]` | U | 1/2 | **1/2** |

The table above shows that after the fix all the properties evaluate to the correct value of 1/2.

*4) Summary:* The Micropay 1 protocol is an example of the smart contract with vulnerability that follows from the nature of blockchain and the way how transactions are handled in Ethereum. Such vulnerabilities are especially hard to spot by hand but they are a perfect example to be verified by *EthVer*. It must be stressed that *EthVer* does not contain a special mechanism implemented to verify the contract against the *front-running attack* or against any other particular attack type. Instead, thanks to the proper modeling of the blockchain, it is able to find any contract bug related to the blockchain and to the wrong handling of the transactions in the network.