Model-Checking of Smart Contracts

Zeinab Nehai, Pierre-Yves Piriou, Frédéric Daumas

▶ **To cite this version:**

Zeinab Nehai, Pierre-Yves Piriou, Frédéric Daumas. Model-Checking of Smart Contracts. IEEE International Conference on Blockchain, Jul 2018, Halifax, Canada. hal-02103511

HAL Id: hal-02103511
https://hal.science/hal-02103511

Submitted on 18 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Model-Checking of Smart Contracts

Zeinab Nehai
Ecole Supérieure d'Ingénieur
Léonard de Vinci
Paris-La Défense
Email: zeinab.nehai@devinci.com

Pierre-Yves Piriou
Electricite de France R&D
78400 Chatou
Email: pierre-yves.piriou@edf.fr

Frederic Daumas
Electricite de France R&D
78400 Chatou
Email: frederic.daumas@edf.fr

phase

***Abstract*—DAO attack showed that formal verification of smart contracts is an important issue that should be addressed to prevent irreversible consequences due to design faults activation in Blockchain applications. This paper proposes a modeling method of an Ethereum application based on smart contracts, with the aim of applying a formal method, namely Model-Checking, to verify that the application implementation complies with its specification, formalized by a set of temporal logic propositions. NuSMV tool has been chosen to support this first approach. The proposed model template is shaped by three layers capturing respectively the behavior of Ethereum blockchain, the smart contracts themselves and the execution framework. The approach is illustrated by a case study coming from energy market field.**

***Keywords*—Ethereum, model-checking, computation tree logic, NuSMV.**

I. Introduction

By Saturday, 18th June 2016, the Distributed Autonomous Organization (DAO) has been hacked. The attacker managed to drain more than 3.6 million ether exploiting a flaw in DAO smart contracts source code. A Blockchain application based on smart contracts is definitively a critical system intended to be checked before any implementation, in order to provide safe behavior and avoid security vulnerabilities. This attack could have been prevented with the use of formal methods. Model-checking [1] is a type of formal methods, it has been used primarily in hardware and protocol verification, and now this technique is applied to analyze specifications of software systems with large reachable states spaces. As a result, model-checking is now powerful enough that it is becoming widely used in industry to aid in the verification of newly developed designs, which is particularly desirable for critical systems.

This paper aims to establish a generic modeling method of a Blockchain Ethereum application, in order to apply a model-checking approach on smart contracts and its execution environment, as described in Figure 1. The proposed model is written in NuSMV input language [2] and the properties to check are formalized into temporal logic CTL [3]. The model is three-fold. Firstly the Ethereum Blockchain [4] is modeled as a distributed system managing transactions between clients. Secondly, modeling rules are proposed to translate the smart contracts from Solidity (the smart contract coding language for Ethereum) into the NuSMV input language. Finally the execution environment has to be tuned according to the intended use case. To check whether the contracts behave as they are supposed to do, expected properties of the application have to be formalized into temporal logic. If a property does not hold then model-checking provides a counter-example, thus it is possible to determine the nature and source of the defect. The method is quite generic and can be applied to various Ethereum applications. The method is applied on a case study coming from energy market field: a Blockchain Energy Market Place.

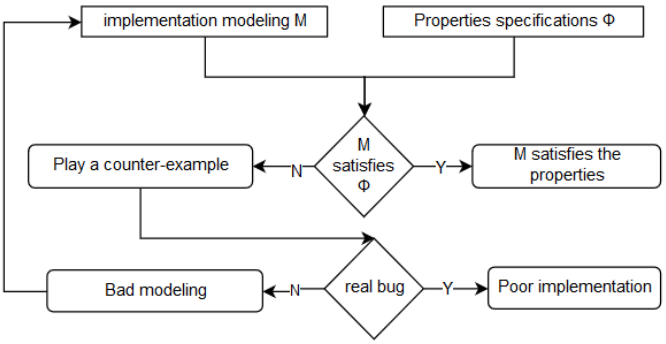

Fig. 1: Model-Checking approach

Because of the novelty of Blockchain technology, only few research has been undertaken on formal verification of smart contracts. Tezos is a Blockchain that intends to run smart contracts, while being able to formally verify the validity of the code. Tezos white paper [5] provides interesting guidelines on the value of formal verification applied to smart contracts, but their concept is to integrate a proof checker within protocols. Our approach does not rely on this restriction. A study done in 2015 [6] focused his attention on the validation of the behavior of the stakeholders when carrying out a contract by combining game theory and formal verification. Our approach aims to verify that smart contracts comply with their specifications for a given behavior of stakeholders. The paper [7] proposes a tool for analyzing and verifying the functional accuracy of Solidity contracts, both at the source level (functional correction specifications) and bytecode level (low-level properties). This approach is the good one to verify an appplication ready to be deployed in the blockchain but seems to be to tedious to apply early in the design process. The present work takes advantage of model-checking technique to fill this lack.

The paper is organized as follows. Section 2 introduces basic notions of Model-Checking. The modeling method is described in section 3, and the case study is addressed in section 4. Finally section 5 provides conclusion remarks and suggests perspectives. For information, the paper uses a Blockchain Ethereum vocabulary, it will be assumed that the reader will

be familiar with this jargon.

II. Recall on Model-Checking

A. Overall description

Model checking is an automatic formal verification technique based on a description of the behavior under study into a state machine [8]. This technique consists in performing an efficient systematic inspection of all possible state sequences described by the model in order to prove it satisfies some behavioral properties. The semantics of a state machine is given by a system of transitions what can be more or less complex, ranging from finite state machines (finite automata) to real programs (Turing machines). Thus, the main challenge in model checking is the combinatory explosion of the model. Nevertheless, this technique is relevant to check partial specifications early in the design process [8]. If the model does not satisfy a property under consideration, the model checker provides a counter-example of sequence that violates the property. This information can be advantageously used to adapt the design (or the specification). Indeed, the wide number of successful industrial applications witnesses the performance of model checking tools [9]. Among other existing tools, SPIN [10], ProB [11], UPPAAL [12] and NuSMV [2] can be cited to represent three kind of model-checkers based on different modelling techniques [13]. This paper proposes a method to build a NuSMV model of a Blockchain application based on smart contracts. NuSMV is widely used for academic research thus is well suited for a first approach.

B. Temporal logic

Properties to be checked on transition system can be various kinds [14]:

- reachability: *"is it possible to end up in a given state?"*
- safety *"something bad never happens"*
- liveness *"something good will eventually happen"*
- fairness *"does, under certain conditions, an event occur repeatedly?"*
- functional correctness: *"does the system do what it is supposed to do?"*
- real-time properties *"is the system acting in time?"*

These properties can be formalized by temporal logic propositions. Such logic define temporal operators[1] in top of classical logic operator (conjonction $\wedge$, disjonction $\vee$, negation !, implication $\rightarrow$, equivalence $\leftrightarrow$). In this paper, we will focus on a temporal logic called Computation Tree Logic (CTL) [3], which is accepted by NuSMV. CTL formulas describe properties of a computation tree, which represents all the possible executions starting from the initial state of the behavior under study. A CTL formula connects atomic properties (evaluable by Booleans) with usual logic operators, two path quantifiers and four temporal operators (three unary and one binary). Given a property p, a path quantifier specifies if p is verified for all paths (Ap) or for at least one (Ep). Temporal operators describes in what states of a path a property is verified:

- Xp (next): p holds in the second state of the path.
- Fp (eventually): p holds in at least one state of the path.
- Gp (always): p holds in every state of the path.
- $p\ Ug$ (until): g holds in at least one state of the path and p holds in every previous states.

III. Modeling a Blockchain application into NuSMV input language

This section sketches a methodology to build a NuSMV model of a Blockchain application based on Ethereum Smart Contracts. The proposed model is three-fold: the *kernel layer* capture the (Ethereum) Blockchain behavior, the *application layer* models the smart contracts themselves and the *environment layer* determines an execution framework for the application. NuSMV input language allows a modulary and hierarchical modelling process. A module can be instantiated into another one and the top module is called *main* (NuSMV syntax and semantics are detailed in [2]). In the proposed model, the *kernel layer* is a set of modules among them one is a template model of a Blockchain client behavior. An Ethereum smart contract is a particular case of Blockchain client, then the *application layer* is a set of refinements of the client module. Finally, the *environment layer* is the module *main* itself.

A. A minimalist model of Blockchain

For this first approach, we limit the Blockchain apprehension to its very basics: a distributed system managing *transactions* between *clients*.

1) MODULE client: An Ethereum Blockchain client, identified by an address, can be either a BC user (an external client) or a smart contract (an internal client). Its state is characterized in our model by two variables (see the code below):

- *balance* (type *integer*) is the number of *wei* (subdivision of the ETHER token) owned by the client. Its maximum value may have a significant impact on analysis calculus time then should be thoroughly chosen in *main* module (refered in the model by the parameter *env*).
- *data* (type *word[64]*[2]) is an arbitrary data stored by the client.

A transition relation states that if the client is not involved in a transaction in current state, then its balance will remain unchanged in next state and its data is reinitiated. This condition is translated by the boolean *will_move* whose expression depends on the analysed scenario then has to be defined in *main* module.

```
MODULE client(env,init_balance)
  VAR
    balance : 0..env.INT_MAX;
```

[1]The word "temporal" in this context does not necessary refer to physical time but logic time: only the order of events occurrence are considered.

[2]The length of data variable is limited to 64 bit here but arbitrary length data could be considered through the type *array of word[64]*.

```
    data : word[64];

  ASSIGN
    init(balance) := init_balance;
    init(data) := 0uh64_0;

  TRANS  !will_move -> conserve ;

  DEFINE
    conserve := (next(balance) = balance)
              & (next(data) = 0h64_0);
```

2) MODULE transaction: A client (the *snder*) can send an *amount* of wei and some *data* to another client (the *rcver*). Such transaction occurs when a parametrable condition becomes true (*!fired & cond*) if the sender owns enough wei (an additional condition on receiver's balance ensures this variable will not overflow). If these conditions are verified (alias *trigger*), the sender and receiver balance and data will be updated in next state.

```
MODULE transaction(env,cond,snder,rcver,
                   amount,data)
  VAR fired : boolean;

  ASSIGN
    init(fired) := FALSE;
    next(fired) := case
      trigger : TRUE;
      ! next(cond) : FALSE;
      TRUE : fired;
      esac;

  TRANS trigger -> (next(snder.data)=0h64_0)
& (next(snder.balance) = snder.balance-amount)
& (next(rcver.data) = resize(data,64))
& (next(rcver.balance) = rcver.balance+amount);

  DEFINE trigger := (!fired & cond
& (snder.balance >= amount)
& (env.INT_MAX - amount >= rcver.balance));
```

In order to illustrate a basic use of this kernel layer, module *main* hereinbelow depicts a simple scenario involving two clients (*c1* and *c2*) who make transactions with an arbitrary amount and data. Both starts with a balance of 10 weis, and we can for instance check that the sum of their balance always equals 20 (which is clearly true). Let us remark that the booleans *will_move* of each instantiated clients are defined in a systematic way: logical OR of all transactions' *trigger* attributes whose the client is either the sender or the receiver.

```
MODULE main
  VAR
    _amount : 0..INT_MAX;
    _data : word[64];
    _cond1 : boolean;
    _cond2 : boolean;
    c1 : client(self,10);
    c2 : client(self,10);
    tx1 : transaction(self,_cond1 &
          !_cond2,c1,c2,_amount,_data);
    tx2 : transaction(self,_cond2 &
          !_cond1,c2,c1,_amount,_data);

  DEFINE
    INT_MAX := 10;
    c1.will_move := tx1.trigger | tx2.trigger;
    c2.will_move := tx1.trigger | tx2.trigger;

SPEC AG (c1.balance + c2.balance = 20);
```

This minimalist model is sufficient to check the smart contracts implementation. To address more ambitious properties on the behavior of an application over time, it would be necessary to refine the model and in particular develop the notions of blocks, ledger, miner, gas consumption...

B. Smart Contract modeling

A smart contract is a particular case of programmable client that automatically reacts when it receives a transaction carrying the appropriate data. Solidity is the mostly used language to develop smart contracts for Ethereum Blockchains [15]. It is a kind of Object-Oriented language where contracts are objects. Thus a contract is defined with a set of typed variables (or attributes) and a set of functions (or methods). A contract instance stores in the BC its attributes values what may be changed when its methods are called. A Solidty developer can also define some *events* to log information on the contract state when a method is executed. This information is logged into the transaction's receipt calling the method what is really convenient for the contract monitoring, but such feature is secundary and is not taken into account in the current approach. To call a contract's method, a client has to send (through a transaction) a specific data to the contract, which is the concatenation of (see the section *Application Binary Interface Specification* in Solidity documentation [15] for full details):

- the function selector: first four bytes of the Keccak256 hash of the function signature;
- the argument encoding: a code determined from the arguments values (see [15]), here simplified as an hexadecimal conversion of arguments.

Let us now consider a toy example of Solidity contract:

```
contract Toy_example {
  uint a = 0;
  function foo(bool b) returns (uint) {
    if (b) {
      a=42;
    }
    else {
      a=0;
    }
  }
}
```

The function selector of function *foo* is "45557578" (first four bytes of keccak256("foo(bool)")) and the argument possible values *false* and *true* can respectively be encoded by "0" and "1". Once deployed in the Blockchain, a client can call the method foo with a boolean value by sending to the contract address the data "455575780" or "455575781".

To model a smart contract in a NuSMV module, we propose to apply following rules:

1) To instantiate the *client* module such as the contract module inherits from a basic client behavior (a contract is a particular case of client).
2) To translate the contract attributes into NuSMV variables according to the table of types equivalences (cf. Table I).
3) To define, for each function f, an alias $call\_f$ for the test: does the stored data begins by the function selector corresponding to f? When the function has arguments, their values also have to be captured into aliases. When the function returns a value, another alias f_return may be created to translate it.
4) To describe with assignments the evolution of variables when functions are executed.
5) To define a transaction when an external function (i.e. a function located in another contract) is called in a local function. The condition to trigger the transaction must capture the calling context.

This list of rules is sufficient to capture the behavior of simple smart contracts and should be refined to address more complex ones. Nevertheless, the limitations of NuSMV language prevent to propose a systematic set of rules, thus subtile constructions should be modelled by specific tricks. The proposed list of rules has the advantage to be fully automatizable.

Solidity	NuSMV
Unsigned integer	Integer [min integer..max integer]
Address	Unsigned word[N] (N: natural number)
Boolean	Boolean
Mapping	Decompose the mapping
Structure	Decompose the structure
Array of <type>	array[min array size]..[max array size] of [type with size]
Enumeration	enumeration type

TABLE I: Type translation from Solidity to NuSMV

The NuSMV code here in below shows the result of the application of these rules to the contract given in example above:

```
MODULE Toy_example(env)
  VAR
    c: client(env,0);
    a: 0..env.INT_MAX;

  ASSIGN
    init(a) := 0;
    next(a) := case
      next(call_foo & b) : 42;
      next(call_foo & !b) : 0;
      TRUE : a;
      esac;

  DEFINE
    call_foo := (c.data[32:1]=0h_45557578);
    b := bool(c.data[0:0]);
```

A possible test case is an oracle that reacts on the updating of a particular boolean *_b* by calling the method *foo*. On this scenario, we can check that the property *"whenever **_b** is true, the attribute **a** stored in the smart contract is equal to 42."* holds for all paths. This test case is modeled by the following main module:

```
MODULE main
  VAR
    _b : boolean;
    toy : Toy_example(self);
    oracle : client(self,0);
    tx : transaction(self,_cond, oracle,
         toy.c,0,foo_selector::word1(_b));

  ASSIGN init(_b):=FALSE;
  TRANS tx.fired -> _b=next(_b);

  DEFINE
    INT_MAX := 100;
    oracle.will_move := tx.trigger;
    toy.c.will_move := tx.trigger;
    foo_selector := 0h_45557578;
    _cond := (_b != next(_b));

SPEC AG (_b -> (toy.a=42));
```

IV. The Blockchain Energy Market Place (BEMP) use case

This section illustrates the approach with a case study: the Blockchain Energy Market Place (BEMP). This Blockchain use case based on smart contracts, popularized by the Brooklyn microgrid [16], aims at managing peer-to-peer energy exchanges between prosumers (productor/consumer) in a microgrid.

A. Description of a BEMP implementation

EDF R&D has developped a proof-of-work implementation of the BEMP use case. Figure 2 depicts an energy trade between two users of the BEMP. In this Figure, transfers 1 and 2 are performed continuously and independently of the market, whereas transfers 3,4 and 5 are performed regularly by smart contracts according to transfer 1 and 2 (an oracle connected to users' smart meters feeds the BEMP with production and consumption data):

1) Alice produces energy and supplies its excess production to the grid.
2) Bob consumes a certain amount of energy pulling from the grid.
3) Alice's production is capitalized as "crypto-kilowatthours" (crypto-kWh) sent to the BEMP.
4) Bob pays Alice with ETHER currency for her energy (the amount to transfer is determined by the BEMP according to her production and his consumption).
5) Bob receives crypto-kWh in proportion of its payment as bill.

The considered implementation is composed of four smart contracts coding into Solidity language:

- ***Account*** which stores the properties of an account (user location, sales and purchasing capacity...) and performs the ETHER payment (one such instance is created for each user).
- ***Market*** which receives all sales and orders and ensure the transit of crypto-kWh.
- ***Algorithm*** which determines the best way to satisfy sales and orders.

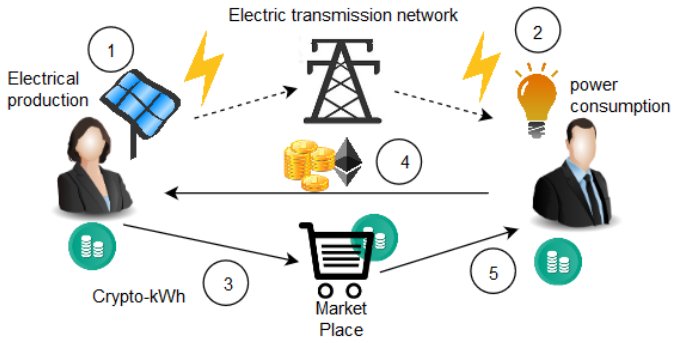

Fig. 2: Operation of an energy and token transaction of *BEMP*

- ***Registry*** that logs any events occuring in BEMP (account creation, sales and orders, actual transactions).

Once deployed in an Ethereum Blockchain, the calling of these contracts' functions is performed by a JavaScript oracle that is running regularly. In practice, the oracle is connected to the Blockchain as a casual client and sends regularly the same sequence of transaction to smart contracts:

1) Opening the market (transaction sending to *Market*);
2) Recording production and consumption amounts of each user (transactions sending to *Registry* contract that triggers three automatic transactions between *Registry*, *Account* and *Market*);
3) Running the algorithm (transaction sending to *Algorithm* that triggers four internal transactions between *Algorithm*, *Registry*, two instances of *Account* and *Market*);
4) Closing the market (transaction sending to *Market*).

B. Modelling the BEMP in NuSMV

Our approch is now applied to check that the presented BEMP implementation verifies some required properties. First the minimalist model of Blockchain introduced in subsection III-A is reused without modification. Then, modeling rules described in subsection III-B have been applied (with some local specific adaptations) to translate the four Solidity smart contracts into NuSMV modules. Finally, the sequence of transactions sending by the oracle has been reproduced into the main module (the scope of the analysis is limited to a single execution of the sequence).

For practical reasons we cannot report here the Solidity code, neither the complete NuSMV model. Only an extract is reported below. This extract show the part of the model concerned by the third step of the sequence performed by the oracle. To illustrates how to use this partial model, we propose a main module that describes a context involving three users: one producer (Alice) and two consumers (Bob and Claude).

```
MODULE main
  FROZENVAR
    Alice_meter : word[64];
    Bob_meter : word[64];
    Claude_meter : word[64];
    export_Alice : word[4] ;
    import_Bob : word[4] ;
    import_Claude : word[4] ;

  VAR
    -- instantiation of clients:
    market : Market(self);
    registry : Registry(self);
    algo : Algorithm(self);
    Alice : Account(self,AliceInitBalance);
    Bob : Account(self,BobInitBalance);
    Claude : Account(self,ClaudeInitBalance);
    oracle : client(self,0); -- external client

    stepper : {s1,s2a,s2b,s2c,s3,s4,end} ;

    -- instantiation of initiator transactions:
    -- calling of method 'openMarket'
    step1 : transaction(self, stepper=s1,
oracle, market.cl, 0, 0uh64_35caf43f);
   -- calling of 'recordImportsAndExports'
   -- method with its parameters: 'meterId',
   -- 'imports' and 'exports' (3 times)
    step2a : transaction(self, stepper=s2a,
oracle, registry.cl, 0, 0uh32_35c768a1::
Alice_meter::0uh8_0::extend(export_Alice,64));
    step2b : transaction(self, stepper=s2b,
oracle, registry.cl, 0, 0uh32_35c768a1::
Bob_meter::extend(import_Bob,64)::0uh8_0);
    step2c : transaction(self, stepper=s2c,
oracle, registry.cl, 0, 0uh32_35c768a1::
Claude_meter::extend(import_Claude,64)::0uh8_0);
   -- calling of 'run' method
    step3 : transaction(self, stepper=s3,
oracle,algo.cl,0,0uh64_83628047);
   -- calling of 'closeMarket' method
    step4 : transaction(self, stepper=s4,
oracle, market.cl, 0, 0uh64_6e155755);

  ASSIGN
    init(stepper) := s1;
    next(stepper) := case
step1.trigger : s2a ;
market.transferToMarket.trigger : s2b ;
Bob.buy.trigger : s2c ;
Claude.buy.trigger : s3 ;
market.transferFromMarket.trigger : s4;
step4.trigger : end ;
TRUE : stepper ;
esac ;

  DEFINE
    INT_MAX := toint(0uh_ff);
AliceInitBalance := 0;
BobInitBalance := 100;
ClaudeInitBalance := 100;
pricePerEnergyUnit := 0d8_10;

    -- Definition of will_move variables
    -- just necessary for this illustration:
    market.cl.will_move :=
step1.trigger |
step5.trigger |
market.transferFromMarket.trigger |
algo.complete1.trigger |
algo.complete2.trigger |
market.completeBob.trigger |
market.completeClaude.trigger;
    registry.cl.will_move :=
```

```
step3a.trigger |
step3b.trigger |
step3c.trigger |
market.transferFromMarket.trigger |
    algo.cl.will_move :=
step4.trigger |
algo.complete1.trigger |
algo.complete2.trigger ;
    Alice.cl.will_move :=
Bob.pay.trigger |
Claude.pay.trigger ;
    Bob.cl.will_move :=
market.completeBob.trigger |
Bob.pay.trigger  ;
    Claude.cl.will_move :=
market.completeClaude.trigger |
Claude.pay.trigger ;
    oracle.will_move :=
step1.trigger |
step3a.trigger |
step3b.trigger |
step3c.trigger |
step4.trigger |
step5.trigger ;
```

This module declares 6 frozen variables to set at initialization a 64 bits word encoding the users' smart meters identifiers, and a 4-bits word encoding the amount of energy they export (for Alice) or import (for Bob and Claude) to/from the microgrid[3]. This data is sent by the users' smart meters. Then the models of smart contracts are instantiated (Account is instantiated three times: one for each user). The oracle is also instantiated as a standard blockchain's client. Then 6 transactions are declared that corresponds to the initiator of the four steps of the sequence executed by the oracle (these transactions are therefore sent by the oracle). The step 2 is repeated 3 times because there are three users for this case. The conditions to trigger each of these transactions are managed by the variable *stepper* whose value change when a step is finished. First and last steps consist in sending a unique transaction, then they are finished as soon as this transaction has been triggered (in the next state). But the other steps trigger a sequence of transactions then they are finished as soon as the last transaction has been triggered. Finally, aliases are declared. In particular, the *will_move* aliases of each client are defined here (cf. subsection III-A).

The initiator transaction of third step aims to call the *run* method of *algo* smart contract. Let us now see how this method is modeled into the module Algorithm.

```
MODULE Algorithm(env)
  VAR
    cl : client(env,0);
    buyAmount1 : word[8];
    buyAmount2 : word[8];
    sellAmount : word[8];
    complete1 : transaction(env,cond_complete1,
self.cl,env.market.cl,0,
0uh32_70ea0793::0h8_1::0h8_1::to_buy1::price1);
    complete2 : transaction(env,cond_complete2,
self.cl,env.market.cl,0,
0uh32_70ea0793::0h8_1::0h8_2::to_buy2::price2);
```

[3]To simplify the code, we assume that these values are always strictly positive. This is possible to specify this constraint at simulation stage.

```
ASSIGN
    init(buyAmount1) := 0h8_0;
    next(buyAmount1) := case
next(call_run) : next(env.market.buyOrder[1]) ;
TRUE : buyAmount1 ;
esac ;
    init(buyAmount2) := 0h8_0;
    next(buyAmount2) := case
next(call_run) : next(env.market.buyOrder[2]) ;
TRUE : buyAmount2 ;
esac ;
    init(sellAmount) := 0h8_0;
    next(sellAmount) := case
next(call_run) : next(env.market.sellOrder[1]) ;
TRUE : sellAmount ;
esac ;

  DEFINE
    call_run := (cl.data = 0h64_83628047) ;
    to_buy1 :=
buyAmount1 + buyAmount2  >= sellAmount ?
(sellAmount*buyAmount1)/(buyAmount1+buyAmount2):
buyAmount1 ;
    to_buy2 :=
buyAmount1 + buyAmount2  >= sellAmount ?
(sellAmount*buyAmount2)/(buyAmount1+buyAmount2):
buyAmount2 ;
    price1 := to_buy1*env.pricePerEnergyUnit;
    price2 := to_buy2*env.pricePerEnergyUnit;
    cond_complete1 := call_run;
    cond_complete2 := complete1.fired;
```

We can see that when the method *run* is called, two transactions are sequentially sent to *market* for completing the orders stored in the market, calling the method *complete*. The implemented algorithm consists in satisfying the users in proportion of their importation/exportation. In our case, that means that if Bob and Claude has consumed more energy than Alice has produced, they share her energy in proportion of their own consumption (of course, if Alice has produced enough for supplying both Bob and Claude, they pay only for their own consumption). The energy price is simply determined by multiplicating the amount of energy bought by a constant factor. The method *complete* has 4 arguments: the sell order id, the buy order id, the amount of energy bought and its price. Let us now see how this method is modeled into the module Market.

```
MODULE ETP_Market(env)
  VAR
    cl : account(env,0);
    completeBob : transaction(env,
cond_completeBob, self.cl, env.Bob.cl,
0, 0uh40_70ea0793::seller_meter::price);
    completeClaude : transaction(env,
cond_completeClaude, self.cl, env.Claude.cl,
0, 0uh40_70ea0793::seller_meter::price);
    transferFromMarket : transaction(env,
cond_transferFrom, self.cl, env.registry.cl,
0, 0uh40_16c5407d::buyer_meter::amount);

  DEFINE
    call_complete := marketOpen &
(cl.data[63:32] = 0h_70ea0793);
    seller := call_complete ?
```

```
cl.data[31:24] : 0h8_0;
    buyer := call_complete ?
cl.data[23:16] : 0h8_0;
    amount := call_complete ?
cl.data[15:8] : 0h8_0;
    price := call_complete ?
cl.data[7:0] : 0h8_0;
    seller_meter := env.Alice_meter;
    buyer_meter := case
cond_completeBob.fired : env.Bob_meter;
cond_completeClaude.fired : env.Claude_meter;
TRUE : 0h16_0;
esac;
    cond_completeBob := call_complete &
buyer = 0h8_1;
    cond_completeClaude := call_complete &
buyer = 0h8_2;
    cond_transferFrom :=
env.Bob.payAlice.trigger |
env.Claude.payAlice.trigger ;
```

Aliases *seller*, *buyer*, *amount* and *price* refer to the four parameters of method *complete*. Parameter *buyer* is used to determine who is the current buyer (Bob is the first registered buyer and Claude is the second one)[4]. When the method *complete* is called, the market send a transaction to either Bob or Claude account (the concerned buyer) calling its method *complete* that trigger the payment. This method has two parameters: the seller identifier and the price to pay. If the buyer actually pay, the market send another transaction to the registry calling its method *transferFromMarket* that edits the buyer's bill. This second method has also two parameters: the buyer identifier and the amount of energy bought. Let us now see how the method *complete* is modeled into the module Account.

```
MODULE Account(env,init_balance)
  VAR
    cl : client(env,init_balance);
    payAlice : transaction(env,cond_payAlice,
self.cl, env.Alice.cl,_price,0uh64_0);

  DEFINE
    call_complete:=(cl.data[55:24]=0h_70ea0793);
    sellerId := call_complete ?
cl.data[15:0] : 0h16_0;
    price := call_complete ?
cl.data[23:16] : 0h8_0;
    _price := toint(price);
    cond_payAlice := call_complete &
(sellerId = env.Alice_meter);
```

When the method *complete* is called, if the seller is Alice, the account send a transaction to Alice with no data but an amount of wei (*_price*). This transaction will succeed only if the account owns enough weis in its balance. Finally, let us see how the method *transferFromMarket* is modeled into the module Registry.

```
MODULE ETP_Registry(env)
  VAR
    ac : account(env,0);
    billOfBob : 0..env.INT_MAX;
    billOfClaude : 0..env.INT_MAX;

  ASSIGN
    init(billOfBob) := 0;
    next(billOfBob) := case
next((buyer_meter = env.Bob_meter)) &
(billOfBob + next(_amount) <= env.INT_MAX) :
billOfBob + next(_amount);
TRUE : billOfBob;
esac;
-- similar assignments here for billOfClaude

  DEFINE
    call_transferFromMarket :=
(ac.data[39:8] = 0h_16c5407d);
    buyer_meter := call_transferFromMarket ?
ac.data[23:8] : 0h16_0;
    amount := call_transferFromMarket ?
ac.data[7:0] : 0h8_0;
    _amount := toint(amount);
```

When the method *transferFromMarket* is called, the bill of either Bob or Claude is updated with the amount of energy bought.

C. Formalizing the BEMP specifications

In this subsection, we introduce five examples of properties that illustrates possible requirements of a BEMP. Such properties are formalized into CTL logic.

Property 1: *If Bob's payment has been achieved, then his bill is edited before the market closure.*

AG (Bob.pay.fired $\longrightarrow$
A [market.marketOpen **U** (registry.billOfBob > 0)]) ;

Property 2: *Alice cannot sell more energy that the amount she has supplied to the grid.*

AG (Alice.cl.balance $\leq$ AliceInitBalance +
toint(export_Alice*pricePerEnergyUnit)) ;

Property 3: *Once opened, the market will eventually be closed (underlying: the algorithm always find a solution).*

AG (Market.marketOpen $\longrightarrow$ **AF** !Market.marketOpen);

Property 4: *The algorithm implements a proportional repartition of energy.*

AG (step3.trigger $\longrightarrow$ **AX** ((algo.to_buy1 * buy_amount2) / (algo.to_buy2 * buy_amount1) $\leq$ 1));

Property 5: *The algorithm implements an equal repartition of energy: if both consumers consume more than the half of the energy produces, then they share it equally.*

AG ((step3.trigger & (import_Bob > export_Alice/2) & (import_Claude > export_Alice/2)) $\longrightarrow$
AX (algo.to_buy1 = algo.to_buy2));

Model-checking can be applied on our model to verify that the considered implementation satisfies the first four properties. However the last one is not satisfied (actually property 4 and 5 are incompatibles in the general case). Model-checking provides a counter example that is shown by Figure 3. We can see that in state 12: algo.to_buy1 $\neq$ algo.to_buy2.

[4]In our case, Alice is the unique seller

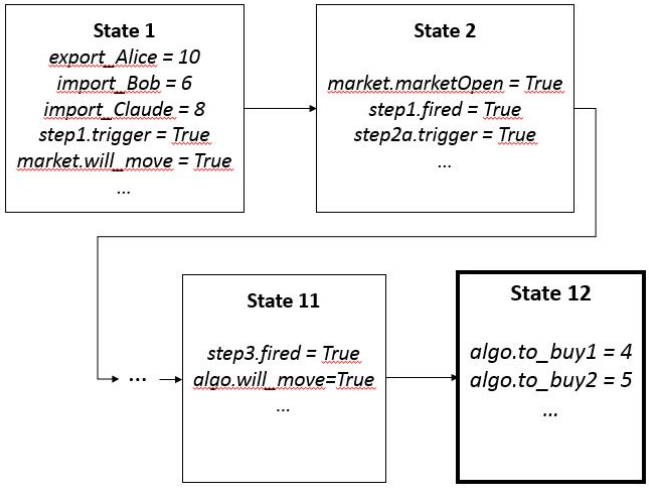

Fig. 3: Counter example showing that the considered implementation does not satisfy property 5

V. Conclusion

This paper present a way of applying model-checking to a Blockchain Ethereum application based on smart contracts. The corner stone of the approach is to build up a model according to a three-fold modeling process, namely the *kernel layer*, the *application layer* and the *environment layer*. Translation rules from Solidity to NuSMV language have been provided to build the application layer. This approach has been applied to a case study coming from the energy market field: the Blockchain Energy Market Place. Finally Model-Checking technique has been exercised on the resulted NuSMV model to assess some properties of interest formalized in CTL logic.

To address more ambitious verification and validation issues, a more precise modelling of an actual Blockchain application is desirable. Such precision cannot be reached by a NuSMV model because of the limitations of its input language. Moreover, the presented approach is limited to application developped into the Ethereum paradigm. Thus we are looking for a higher level formalism to build a more abstract model of a Blockchain, independant from its main implementation technologies. Such simulable model of a Blockchain and its applicative smart contracts would serve two main complementary purposes:

- functionnal validation of smart contracts at their requirements specification development phase.
- reference for end-verification towards the specifications when integrating their implementations at final system validation phase.

References

[1] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[2] Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Andrei Tchaltsev. Nusmv 2.4 user manual. *CMU and ITC-irst*, 2005.

[3] M.C. Browne, E.M. Clarke, and O. Grümberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1):115 – 131, 1988.

[4] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[5] LM Goodman. Tezos—a self-amending crypto-ledger white paper, 2014.

[6] Giancarlo Bigi, Andrea Bracciali, Giovanni Meacci, and Emilio Tuosto. Validation of decentralised smart contracts through game theory and formal methods. In *Programming Languages with Applications to Biology and Security*, pages 142–161. Springer, 2015.

[7] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Short paper: Formal verification of smart contracts.

[8] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[9] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of model checking*. MIT press, 2008.

[10] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[11] Michael Leuschel, Michael Butler, et al. Prob: A model checker for b. In *FME*, volume 2805, pages 855–874. Springer, 2003.

[12] Kim Guldstrand Larsen, Paul Pettersson, and Wang yi. Uppaal in a nutshell. 1:134–152, 12 1997.

[13] Marc Frappier, Benoît Fraikin, Romain Chossart, Raphaël Chane-Yack-Fa, and Mohammed Ouenzar. Comparison of model checking tools for information systems. In *International Conference on Formal Engineering Methods*, pages 581–596. Springer, 2010.

[14] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[15] Ethereum. Solidity documentation, release 0.4.17, 2017.

[16] Esther Mengelkamp, Johannes Gärttner, Kerstin Rock, Scott Kessler, Lawrence Orsini, and Christof Weinhardt. Designing microgrid energy markets: a case study: the brooklyn microgrid. *Applied Energy*, 2017.