



Deductive Proof of Ethereum Smart Contracts Using Why3

Zeinab Nehai, François Bobot

► To cite this version:

Zeinab Nehai, François Bobot. Deductive Proof of Ethereum Smart Contracts Using Why3. [Research Report] CEA DILS. 2019. hal-02108987v2

HAL Id: hal-02108987

<https://hal.science/hal-02108987v2>

Submitted on 7 Jul 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.

Deductive Proof of Industrial Smart Contracts Using Why3

Zeinab Nehai^{1,2} and François Bobot²

¹ Université Paris Diderot, Paris, France

² CEA LIST, Palaiseau, France

zeinab.nehai@univ-paris-diderot.fr

{zeinab.nehai, francois.bobot}@cea.fr

Abstract. In this paper, we use a formal language that performs deductive verification on industrial smart contracts, which are self-executing digital programs. Because smart contracts manipulate cryptocurrency and transaction information, if a bug occurs in such programs, serious consequences can happen, such as a loss of money. The aim of this paper is to show that a language dedicated to deductive verification, called *Why3*, can be a suitable language to write correct and proven contracts. We first encode existing contracts into the *Why3* program; next, we formulate specifications to be proved as the absence of RunTime Error and functional properties, then we verify the behaviour of the program using the *Why3* system. Finally, we compile the *Why3* contracts to the Ethereum Virtual Machine (EVM). Moreover, our approach estimates the cost of gas, which is a unit that measures the amount of computational effort during a transaction.

Keywords: deductive verification, why3, smart contracts, solidity.

1 Introduction

Smart Contracts [20] are sequential and executable programs that run on Blockchains [17]. They permit trusted transactions and agreements to be carried out among parties without the need for a central authority while keeping transactions traceable, transparent, and irreversible. These contracts are increasingly confronted with various attacks exploiting their execution vulnerabilities. Attacks lead to significant malicious scenarios, such as the infamous *The DAO* attack [7], resulting in a loss of ~\$60M. In this paper, we use formal methods on smart contracts from an existing Blockchain application. Our motivation is to ensure safe and correct contracts, avoiding the presence of computer bugs, by using a deductive verification language able to write, verify and compile such programs. The chosen language is an automated tool called *Why3* [13], which is a complete tool to perform deductive program verification, based on Hoare logic. A first approach using *Why3* on solidity contracts (the Ethereum smart contracts language) has already been undertaken [2]. The author uses *Why3* to formally verify *Solidity* contracts based on code annotation. Unfortunately,

that work remained at the prototype level. We describe our research approach through a use case that has already been the subject of previous work, namely the Blockchain Energy Market Place (BEMP) application [18]. In summary, the contributions of this paper are as follows:

1. Showing the adaptability of *Why3* as a formal language for writing, checking and compiling smart contracts.
2. Comparing existing smart contracts, written in *Solidity* [11], and the same existing contracts written in *Why3*.
3. Detailing a formal and verified *Trading* contract, an example of a more complicated contract than the majority of existing *Solidity* contracts.
4. Providing a way to prove the quantity of *gas* (fraction of an Ethereum token needed for each transaction) used by a smart contract.

The paper is organized as follows. Section 2 describes the approach from a theoretical and formal point of view by explaining the choices made in the study, and section 3 is the proof-of-concept of compiling *Why3* contracts. A state-of-the-art review of existing work concerning the formal verification of smart contracts is described in section 4. Finally, section 5 summarizes conclusions.

2 A New Approach to Verifying Smart Contracts Using *Why3*

2.1 Background of the study

Deductive approach & Why3 tool. A previous work aimed to verify smart contracts using an abstraction method, model-checking [18]. Despite interesting results from this modelling method, the approach to property verification was not satisfactory. Indeed, it is well-known that model-checking confronts us either with limitation on combinatorial explosion, or limitation with invariant generation. Thus, proving properties involving a large number of states was impossible to achieve because of these limitations. This conclusion led us to consider applying another formal methods technique, deductive verification, which has the advantage of being less dependent on the size of the state space. In this approach, the user is asked to write the invariants. We chose the automated *Why3* tool [13] as our platform for deductive verification. It provides a rich language for specification and programming, called *WhyML*, and relies on well-known external theorem provers such as Alt-ergo [10], Z3 [16], and CVC4 [8]. *Why3* comes with a standard library³ of logical theories and programming data structures. The logic of *Why3* is a first-order logic with polymorphic types and several extensions: recursive definitions, algebraic data types and inductive predicates.

³ <http://why3.lri.fr/>

Case study: Blockchain Energy Market Place. We have applied our approach to a case study provided by industry [18]. It is an Ethereum Blockchain application (BEMP) based on *Solidity* smart contracts language. Briefly, this Blockchain application makes it possible to manage energy exchanges in a peer-to-peer way among the inhabitants of a district as shown in Figure 1. The figure illustrates (1) & (1') energy production (Alice) and energy consumption (Bob). (2) & (2') Smart meters provide production/consumption data to Ethereum blockchain. (3) Bob pays Alice in *ether* (Ethereum's cryptocurrency) for his energy consumption. For more details about the application, please refer to [18].

In our initial work, we applied our method on a simplified version of the application, that is, a one-to-one exchange (1 producer and 1 consumer), with a fixed price for each kilowatt-hour. This first test allowed us to identify and prove RTE properties. The simplicity of the unidirectional exchange model did not allow the definition of complex functional properties to show the importance and utility of the *Why3* tool. In a second step, we extended the application under study to an indefinite number of users, and then enriched our specifications. The use of *Why3* is quite suitable for this order of magnitude. In this second version, we have a set of consumers and producers willing to buy or to sell energy. Accordingly, we introduced a simple trading algorithm that matches producers with consumers. In addition to transferring *ether*, users transfer crypto-Kilowatthours to reward consumers consuming locally produced energy. Hence, the system needs to formulate and prove predicates and properties of functions handling various data other than cryptocurrency. For a first trading approach, we adopted, to our case study, an order book matching algorithm [12].

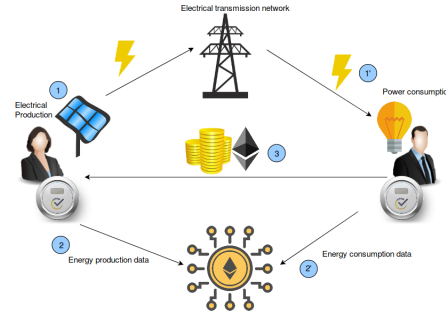


Fig. 1. BEMP Process

2.2 Why3 features intended for Smart Contracts

Library modelling. *Solidity* is an imperative object-oriented programming language, characterized by static typing⁴. It provides several elementary types that can be combined to form complex types such as booleans, signed, unsigned, and fixed-width integers, settings, and domain-specific types like addresses. Moreover, the address type has primitive functions able to transfer *ether* (`send()`, `transfer()`) or manipulate cryptocurrency balances (`.balance`). *Solidity* contains elements that are not part of the *Why3* language. One could

⁴ Ethereum foundation: Solidity, the contract-oriented programming language.
<https://github.com/ethereum/solidity>

model these as additional types or primitive features. Examples of such types are `uint256` and `address`. For machine integers, we use the range feature of Why3: `type uint256 = <range 0 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF... >` because it exactly represents the set of values we want to represent. Moreover why3 checks that the constants written by the user of this types are inside the bounds and converts in specifications automatically range types to the mathematical integers, e.g., `int` type. Indeed it is a lot more natural and clearer to express specification with mathematical integers, for example with wrap-around semantic `account = old account - transfer` doesn't express that the account lose money (if the account was empty it could now have the maximum quantity of money).

Based on the same reasoning, we have modelled the type `Int160`, `Uint160` (which characterizes type `uint` in *Solidity*). We also model the `address` type and its members. We choose to encode the private storage (`balance`) by a Hashtable having as a key value an address, and the associated value a `uint256` value. The current value of the balance of addresses would be `balance[address]`. In addition, the `send` function is translated by a `val` function, which performs operations on the `balance` hashtable. Moreover, we model primitive features such as the `modifier` function, whose role is to restrict access to a function; it can be used to model the states and guard against incorrect usage of the contract. In *Why3* this feature would be an exception to be raised if the condition is not respected, or a precondition to satisfy. We will explain it in more details with an example later. Finally, we give a model of *gas*, in order to specify the maximum amount of *gas* needed in any case. We introduce a new type: `type gas = int`. The quantity of *gas* is modelled as a mathematical integer because it is never manipulated directly by the program. This part is detailed later.

It is important to note that the purpose of our work is not to achieve a complete encoding of *Solidity*. The interest is rather to rely on the case study in our possession (which turns out to be written in *Solidity*), and from its contracts, we build our own *Why3* contracts. Therefore, throughout the article, we have chosen to encode only *Solidity* features encountered through our case study. Consequently, notions like `revert` or `delegatecall` are not treated. Conversely, we introduce additional types such as `order` and `order_trading`, which are specific to the BEMP application. The `order` type is a record that contains `orderAddress` which can be a seller or a buyer, `tokens` that express the crypto-Kilowatthours (wiling to buy or to sell), and `price_order`. The `order_trading` type is a record that contains seller ID; `seller_index`, buyer ID; `buyer_index`, the transferred amount `amount_t`, and the trading price `price_t`.

Remark: In our methodology, we make the choice to encode some primitives of *Solidity* but not all. For example, the `send()` function in *Solidity* can fail (return `False`) due to an out-of-gas, e.g. an overrun of 2300 units of *gas*. The reason is that in certain cases the transfer of *ether* to a contract involves the execution of the contract fallback, therefore the function might consume more *gas* than expected. A fallback function is a function without a signature (no name, no parameters), it is executed if a contract is called and no other function matches the specified function identifier, or if no data is supplied. As we made the choice

of a *private* blockchain type, all users can be identified and we have control on who can write or read from the blockchain. Thus, the *Why3* `send()` function does not need a fallback execution, it only transfers *ether* from one address to another. The *Why3* `send()` function does not return a boolean, because we require that the transfer is possible (enough ether in the sending contract and not too much in the receiving) and we want to avoid Denial-of-service attack [3]. Indeed if we allow to propagate errors and accept to send to untrusted contracts, it could always make our contract fail and revert. So we can't prove any property of progress of our contract. In *Tezos* blockchain [14], call to other contracts are postponed to after the execution of the current contract. So another contract should not be able to make the calling contract fail.

Encoding and verifying functions from the BEMP application.

Oracle notions. Developing smart contracts often rely on the concept of *Oracles* [1]. An oracle can be seen as the link between the blockchain and the “real world”. Some smart contracts functions have arguments that are external to the blockchain. However, the blockchain does not have access to information from an off-chain data source which is untrusted. Accordingly, the oracle provides a service responsible for entering external data into the blockchain, having the role of a trusted third party. However, questions arise about the reliability of such oracles and accuracy of information. Oracles can have unpredictable behaviour, e.g. a sensor that measures the temperature might be an oracle, but might be faulty; thus one must account for invalid information from oracles. Figure 2 illustrates the three communication stages between various systems in the real world with the blockchain: (1) the collection of off-chain raw data; (2) this data is collected by oracles; and finally, (3) oracles provide information to the blockchain (via smart contracts).

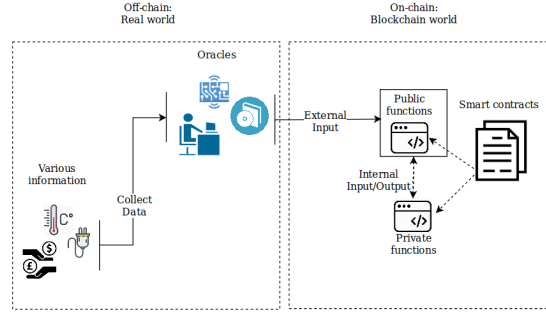


Fig. 2. Link between on-chain and off-chain

Based on this distinction, we defined two types of functions involved in contracts, namely *Private functions* and *Public functions*. We noted that some functions are called internally, by other smart contracts functions, while others are called externally by oracles. Functions that interact with oracles are defined as *public* functions. The proof approach of the two types is different. For the *private* functions one defines pre-conditions and post-conditions, and then we prove that no error can occur and that the function behaves as it should. It

is thus not necessary to define exceptions to be raised throughout the program; they are proved to never occur. Conversely, the *public* functions are called by oracles, the behaviour of the function must, therefore, take into account any input values and it is not possible to require conditions upstream of the call. So in contrast, the exceptions are necessary; we use so-called *defensive proof* in order to protect ourselves from the errors that can be generated by oracles. No constraints are applied on post-conditions. Thus, valid data (which does not raise exceptions) received by a public function will satisfy the pre-conditions of the public function that uses it, because pre-conditions are proved.

Methodology of proving BEMP functions. To illustrate our methodology, we take an example from BEMP.

```

1 function transferFromMarket(address _to, uint _value) onlyMarket returns (
    bool success) {
2     if (exportBalanceOf[market] >= _value)
3     { /* Transferring _value from market to _to */
4     else {success = false;
5         Error("Tokens couldn't be transferred from market");}}
```

The function allows transferring `_value` (expressing cryptokwh) from the `market` to `_to` address. The mapping `exportBalanceOf []` stores balances corresponding to addresses that export tokens. The function can be executed solely by the market (the modifier function `onlyMarket`). The program checks if the market has enough tokens to send to `_to`. If this condition is verified, then the transfer is done. If the condition is not verified, the function returns `false` and triggers an `Error` event (a feature that allows writing logs in the blockchain)⁵. This process is internal to the blockchain, there is no external exchange, hence the function is qualified as *private*. According to the modelling approach, we define complete pre-conditions and post-conditions to verify and prove the function. The corresponding *Why3* function is:

```

1 let transferFromMarket (_to : address) (_value : uint) : bool
2   requires {!onlymarket ^ _value > 0 }
3   requires {marketBalanceOf[market] ≥ _value }
4   requires {importBalanceOf[_to] ≤ max_uint - _value}
5   ensures {(old marketBalanceOf[market]) + (old importBalanceOf[_to]) = marketBalanceOf[
6     market] + importBalanceOf[_to]}
   = (* The program *)
```

The pre-condition in line 2 expresses the modifier `onlyMarket` function. Note that `marketBalanceOf` is the hashtable that records crypto-Kilowatthours balances associated with market addresses, and `importBalanceOf` is the hashtable that records the amount of crypto-Kilowatthours intended for the buyer addresses. From the specification, we understand the behaviour of the function without referencing to the program. To be executed, `transferFromMarket` must respect RTE and functional properties:

⁵ <https://media.consensys.net/technical-introduction-to-events-and-logs-in-ethereum-a074d65dd61e>

- RTE properties: (1) *Positive values*; a valid amount of crypto-Kilowatthours to transfer is a positive amount (Line 2). (2) *Integer overflow*; no overflow will occur when `_to` receives `_value` (Line 4).
- Functional properties: (1) *Acceptable transfer*; the transfer can be done, if the market has enough crypto-Kilowatthours to send (Line 3). (2) *Successful transfer*; the transaction is completed successfully if the sum of the sender and the receiver balance before and after the execution does not change (Line 5). (3) *modifier function*; the function can be executed only by the market (Line 2).

The set of specifications is necessary and sufficient to prove the expected behaviour of the function.

The following function illustrates a *Solidity* public function.

```

1 function registerSmartMeter(string _meterId, address _ownerAddress) onlyOwner
  { addressOf[_meterId] = _ownerAddress;
2    MeterRegistered(_ownerAddress, _meterId);}

```

The function `registerSmartMeters()` is identified by a name (`meterID`) and an owner (`ownerAddress`). Note that all meter owners are recorded in a hashtable `addressOf` associated with a key value `meterID` of the `string` type. The main potential bug in this function is possibly registering a meter twice. When a meter is registered, the function broadcasts an event `MeterRegistered`. Following the modelling rules, there are no pre-conditions, instead, we define exceptions. The corresponding *Why3* function is:

```

1 Exception OnlyOwner, ExistingSmartMeter
2 let registerSmartMeter (meterID : string) (ownerAddress : address)
3   raises { OnlyOwner → !onlyOwner = False }
4   raises { ExistingSmartMeter → mem addressOf meterID }
5   ensures { (size addressOf) = (size (old addressOf) + 1) }
6   ensures { mem addressOf meterID }
7   = (*The program*)

```

The first exception (Line 3) is the *modifier function* which restricts the function execution to the owner, the caller function. It is not possible to pre-condition inputs of the function, so we manage exceptional conditions during the execution of the program. To be executed, `registerSmartMeter` must respect RTE and functional properties:

- RTE properties: *Duplicate record*; if a smart meter and its owner is recorded twice, raise an exception (Line 4)
- Functional properties: (1) *modifier function*; the function can be executed only by the owner, thus we raise `OnlyOwner` when the caller of the function is not the owner (Line 3). (2) *Successful record*; at the end of the function execution, we ensure (Line 5) that a record has made. (3) *Existing record*; the registered smart meter has been properly recorded in the hashtable `addressOf` (Line 6).

The set of specifications is necessary and sufficient to prove the expected behaviour of the function.

Trading contract. The trading algorithm allows matching a potential consumer with a potential seller, recorded in two arrays `buy_order` and `sell_order` taken as parameters of the algorithm. In order to obtain an expected result at the end of the algorithm, properties must be respected. We define specifications that make it possible throughout the trading process. The algorithm is a private function type because it runs on-chain. Thus no exceptions are defined but pre-conditions are. The Trading contract has no *Solidity* equivalent because it is a function added to the original BEMP project. Below is the set of properties of the function:

```

1 let trading (buy_order : array order) (sell_order : array order) : list order_trading
2   requires { length buy_order > 0 ∧ length sell_order > 0 }
3   requires {sorted_order buy_order}
4   requires {sorted_order sell_order}
5   requires {forall j:int. 0 ≤ j < length buy_order → 0 < buy_order[j].tokens }
6   requires {forall j:int. 0 ≤ j < length sell_order → 0 < sell_order[j].tokens }
7   ensures { correct result (old buy_order) (old sell_order) }
8   ensures { forall l. correct l (old buy_order) (old sell_order) →
              nb_token l ≤ nb_token result }
9   ensures {!gas ≤ old !gas + 374 + (length buy_order + length sell_order) * 363}
10  ensures {!alloc ≤ old !alloc + 35 + (length buy_order + length sell_order) * 35}
11  = (* The program *)

```

- RTE properties: *positive values*; parameters of the functions must not be empty (empty array) (Line 2), and a trade cannot be done with null or negative tokens (Lines 5, 6).
- Functional requirements: *sorted orders*; the orders need to be sorted in a decreasing way. Sellers and buyers asking for the most expensive price of energy will be at the top of the list (Lines 3, 4).
- Functional properties: (1) *correct trading* (Lines 7, 8); for a trading to be qualified as correct, it must satisfy two properties:
 - the conservation of buyer and seller tokens that states no loss of tokens during the trading process : `forall i:uint. 0 ≤ i < length sell_order → sum_seller (list_trading) i ≤ sell_order[i].tokens`. For the buyer it is equivalent by replacing seller by buyer.
 - a successful matching; a match between a seller and a buyer is qualified as correct if the price offered by the seller is less than or equal to that of the buyer, and that the sellers and buyers are valid indices in the array.
- (2) *Best tokens exchange*; we choose to qualify a trade as being one of the best if it maximize the total number of tokens exchanged. Line 8 ensures that no correct trading list can have more tokens exchanged than the one resulting from the function. The criteria could be refined by adding that we then want to maximize or minimize the sum of paid (best for seller or for buyer).
- (3) *Gas consumption*; Lines 9 and 10 ensures that no extra-consumption of gas will happen (see the following paragraph).

Gas consumption proof. Overconsumption of *gas* can be avoided by the *gas* model. Instructions in EVM consume an amount of *gas*, and they are categorized

by level of difficulty; e.g., for the set $W_{verylow} = \{ADD, SUB, \dots\}$, the amount to pay is $G_{verylow} = 3 \text{ units of gas}$, and for a create operation the amount to pay is $G_{create} = 32000 \text{ units of gas}$ [20]. The price of an operation is proportional to its difficulty. Accordingly, we fix for each *Why3* function, the appropriate amount of *gas* needed to execute it. Thus, at the end of the function instructions, a variable `gas` expresses the total quantity of *gas* consumed during the process. We introduce a `val ghost` function that adds to the variable `gas` the amount of *gas* consumed by each function calling `add_gas` (see section 3 for more details on *gas* allocation).

```

1 val ghost add_gas (used : gas) (allocation: int): unit
2   requires { 0 ≤ used ∧ 0 ≤ allocation }
3   ensures { !gas = (old !gas) + used }
4   ensures { !alloc = (old !alloc) + allocation }
5   writes { gas, alloc}

```

The specifications of the function above require *positive values* (Line 2). Moreover, at the end of the function, we ensure that there is no extra *gas* consumption (Lines 3, 4). Line 5 specifies the changing variables.

3 Compiling Why3 Contracts and Proving Gas Consumption

The final step of the approach is the deployment of *Why3* contracts. EVM is designed to be the runtime environment for the smart contracts on the Ethereum blockchain [20]. The EVM is a stack-based machine (word of 256 bits) and uses a set of instructions (called opcodes)⁶ to execute specific tasks. The EVM features two memories, one volatile that does not survive the current transaction and a second for storage that does survive but is a lot more expensive to modify. The goal of this section is to describe the approach of compiling *Why3* contracts into EVM code and proving the cost of functions. The compilation⁷ is done in three phases: (1) compiling to an EVM that uses symbolic labels for jump destination and macro instructions. (2) computing the absolute address of the labels, it must be done inside a fixpoint because the size of the jump addresses has an impact on the size of the instruction. Finally, (3) translating the assembly code to pure EVM assembly and printed. Most of *Why3* can be translated, the proof-of-concept compiler allows using algebraic datatypes, not nested pattern-matching, mutable records, recursive functions, while loops, integer bounded arithmetic (32, 64, 128, 256 bits). Global variables are restricted to mutable records with fields of integers. It could be extended to hashtables using the hashing technique of the keys used in *Solidity*. Without using specific instructions, like for C, *Why3* is extracted to garbage collected language, here all the allocations are done in the volatile memory, so the memory is reclaimed only at the end of the transaction.

⁶ <https://ethervm.io>

⁷ The implementation can be found at <http://francois.bobot.eu/fm2019/>

We have not formally proved yet the correction of the compilation, we only tested the compiler using reference interpreter $\llbracket \cdot \rrbracket$ and by asserting some invariants during the transformation. However, we could list the following arguments for the correction:

- the compilation of why3 (ML-language) is straightforward to stack machine.
- the precondition on all the arithmetic operations (always bounded) ensures arithmetic operations could directly use 256bit operations
- raise accepted only in public function before any mutation so the fact they are translated into revert does not change their semantics. `try with` are forbidden.
- only immutable datatype can be stored in the permanent store. Currently, only integers can be stored, it could be extended to other immutable datatype by copying the data to and from the store.
- The send function in why3 only modifies the state of balance of the contracts, requires that the transfer is acceptable and never fail, as discussed previously. So it is compiled similarly to the solidity function send function with a gas limit small enough to disallow modification of the store. Additionally, we discard the result.

The execution of each bytecode instruction has an associated cost. One must pay some *gas* when sending a transaction; if there is not enough *gas* to execute the transaction, the execution stops and the state is rolled back. So it is important to be sure that at any later date the execution of a smart contract will not require an unreasonable quantity of *gas*. The computation of WCET is facilitated in EVM by the absence of cache. So we could use techniques of [6] which annotate in the source code the quantity of *gas* used, here using a function `add_gas` used `allocations`. The number of allocations is important because the real *gas* consumption of EVM integrates the maximum quantity of volatile memory used. The compilation checks that all the paths of the function have a cost smaller than the sum of the `add_gas` `g` `a` on it. The paths of a function are defined on the EVM code by starting at the function-entry and loop-head and going through the code following jumps that are not going back to loop-head.

```

1 let rec mk_list42 [0 evm:gas_checking] (i:int32) : list int32
2 requires { 0 ≤ i } ensures { i = length result } variant { i }
3 ensures { !gas - old !gas ≤ i * 185 + 113 }
4 ensures { !alloc - old !alloc ≤ i * 96 + 32 } =
5   if i ≤ 0 then (add_gas 113 32; Nil)
6   else (let l = mk_list42 (i-1) in add_gas 185 96; Cons (0x42:int32) l)

```

Currently, the cost of the modification of storage is over-approximated; using specific contract for the functions that modify it we could specify that it is less expensive to use a memory cell already used.

4 Related Work

Since the *DAO* attack, the introduction of formal methods at the level of smart contracts has increased. Raziel is a framework to prove the validity of smart

contracts to third parties before their execution in a private way [19]. In that paper, the authors also use a deductive proof approach, but their concept is based on Proof-Carrying Code (PCC) infrastructure, which consists of annotating the source code, thus proofs can be checked before contract execution to verify their validity. Our method does not consist in annotating the *Solidity* source code but in writing the contract program and thus getting a correct-by-construction program. Another widespread approach is static analysis tools. One of them is called Oyente. It has been developed to analyze Ethereum smart contracts to detect bugs. In the corresponding paper [15], the authors were able to run Oyente on 19,366 existing Ethereum contracts, and as a result, the tool flagged 8,833 of them as vulnerable. Although that work provides interesting conclusions, it uses symbolic execution, analyzing paths, so it does not allow to prove functional properties of the entire application. We can also mention the work undertaken by the F^* community [9] where they use their functional programming language to translate *Solidity* contracts to shallow-embedded F^* programs. Just like [5] where the authors perform static analysis by translating *Solidity* contracts into Java using *KeY* [4]. The initiative of the current paper is directly related to a previous work [18], which dealt with formally verifying the smart contracts application by using model-checking. The paper established a methodology to construct a three-fold model of an Ethereum application, with properties formalized in temporal logic CTL. However, because of the limitation of the model-checker used, ambitious verification could not be achieved (e.g., a model for m consumers and n producers). This present work aims to surpass the limits encountered with model-checking, by using a deductive proof approach on an Ethereum application using the *Why3* tool.

5 Conclusions

In this paper, we applied concepts of deductive verification to a computer protocol intended to enforce some transaction rules within an Ethereum blockchain application. The aim is to avoid errors that could have serious consequences. Reproducing, with *Why3*, the behaviour of *Solidity* functions showed that *Why3* is suitable for writing and verifying smart contracts programs. The presented method was applied to a use case that describes an energy market place allowing local energy trading among inhabitants of a neighbourhood. The resulting modelling allows establishing a trading contract, in order to match consumers with producers willing to make a transaction. In addition, this last point demonstrates that with a deductive approach it is possible to model and prove the operation of the BEMP application at realistic scale (e.g. matching m consumers with n producers), contrary to model-checking in [18], thus allowing the verifying of more realistic functional properties.

References

1. Ethereum foundation : Ethereum and oracles. <https://blog.ethereum.org/2014/07/22/ethereum-and-oracles/>
2. Formal verification for solidity contracts. <https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts>
3. Solidity hacks and vulnerabilities. <https://hackernoon.com/hackpedia-16-solidity-hacks-vulnerabilities-their-fixes-and-real-world-examples-f3210eba5148>
4. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: *Deductive software verification-the key book*. lncs, vol. 10001 (2016)
5. Ahrendt, W., Bubel, R., Ellul, J., Pace, G.J., Pardo, R., Rebiscoul, V., Schneider, G.: *Verification of smart contract business logic* (2019)
6. Amadio, R.M., Ayache, N., Bobot, F., Boender, J.P., Campbell, B., Garnier, I., Madet, A., McKinna, J., Mulligan, D.P., Piccolo, M., Pollack, R., Régis-Gianas, Y., Sacerdoti Coen, C., Stark, I., Tranquilli, P.: *Certified complexity (cerco)*. In: Dal Lago, U., Peña, R. (eds.) *Foundational and Practical Aspects of Resource Analysis*. pp. 1–18. Springer International Publishing, Cham (2014)
7. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts. In: *Principles of Security and Trust*, pp. 164–186. Springer (2017)
8. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. Springer (2011)
9. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: *Short paper: Formal verification of smart contracts* (2016)
10. Bobot, F., Conchon, S., Contejean, E., Iguernelala, M., Lescuyer, S., Mebsout, A.: *The alt-ergo automated theorem prover* (2008), <http://alt-ergo.lri.fr/>
11. Buterin, V., et al.: *A next-generation smart contract and decentralized application platform*. white paper (2014)
12. Domowitz, I.: A taxonomy of automated trade execution systems. *Journal of International Money and Finance* **12**, 607–631 (1993)
13. Filliâtre, J.C., Paskevich, A.: Why3 – where programs meet provers. In: *European Symposium on Programming*. pp. 125–128. Springer (2013)
14. Goodman, L.: *Tezos: A self-amending crypto-ledger position paper* (2014)
15. 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)
16. de Moura, L., Bjørner, N.: Z3, an efficient SMT solver, <http://research.microsoft.com/projects/z3/>
17. Nakamoto, S.: *Bitcoin: A peer-to-peer electronic cash system* (2008)
18. Nehaï, Z., Piriou, P.Y., Daumas, F.: *Model-checking of smart contracts*. In: *The 2018 IEEE International Conference on Blockchain*. IEEE (2018)
19. Sánchez, D.C.: *Raziel: Private and verifiable smart contracts on blockchains*. Cryptology ePrint Archive, Report 2017/878 (2017), <http://eprint.iacr.org/2017/878.pdf>, accessed:2017-09-26
20. Wood, G.: *Ethereum: A secure decentralised generalised transaction ledger*. Ethereum project yellow paper **151**, 1–32 (2014)

Appendix A : BEMP Application

```

1 module DCC (*the module that materializes the smart meters*)
2   use my_library.Uint
3   use my_library.SmartMeterID
4   use my_library.Address
5   use array.Array
6
7   (*records of potential vendeur and acheteur, with the purchase (price_b
8     ) and sale (price_s) price*)
9   (*amount_b the needed token quantity, and amount_s the token quantity
10     on sale*)
11
12   type pot_buy = {address_b : address;
13                   smb_id: smartMeterID;
14                   price_b: uint;
15                   amount_b: uint}
16
17   type pot_sell = {address_s : address;
18                   sms_id : smartMeterID;
19                   price_s: uint;
20                   amount_s: uint}
21
22   (*buy_array and sell_array are data tables retrieved from the meters
23     *)
24   val buy_array : array pot_buy
25   val sell_array : array pot_sell
26
27 end
28 module Trading
29   use my_library.Uint
30   use int.Int
31   use int.MinMax
32   use seq.Seq
33   use import my_library.ArrayUint as Arr
34   use ref.Refint
35   use list.List
36   use import list.Length as Len
37   use list.NthNoOpt
38   use my_library.SmartMeterID
39   use my_library.Address
40   use list.HdTLNoOpt
41   use list.NthHdTL
42   use list.Nth as Elem
43
44

```

```

45   type order = {orderAddress : address; tokens: uint; price_order: uint
46   } (*It can be buy or sell , tokens = energy materializes in token*)
47
48   clone array.Sorted as Sort with type elt = order
49
50   val sorted_array (a: array order) : unit
51     ensures {forall i j: int. 0 ≤ j ≤ i < Arr.length a → Uint.to_int
52     (a[i].price_order) ≤ Uint.to_int(a[j].price_order)}
53     writes {a}
54
55   predicate sorted_order (a: Seq.seq order) =
56     forall k1 k2 : int. 0 ≤ k1 ≤ k2 < Seq.length a →
57     Uint.to_int(a[k2].price_order) ≤ Uint.to_int(a[k1].price_order)
58   (**)
59
60   type order_trading = {seller_index: uint; buyer_index: uint; amount_t
61   : uint}
62
63   predicate matching_order (k: order_trading) (b_order : Seq.seq order)
64     (s_order : Seq.seq order) =
65     s_order[k.seller_index].price_order ≤
66     b_order[k.buyer_index].price_order ∧
67     0 ≤ k.buyer_index < Seq.length b_order ∧
68     0 ≤ k.seller_index < Seq.length s_order ∧
69     0 < k.amount_t
70
71   predicate matching (order: list order_trading) (b_order : Seq.seq
72   order) (s_order : Seq.seq order) =
73     match order with
74     | Nil → true
75     | Cons k l → matching l b_order s_order ∧
76     matching_order k b_order s_order
77   end
78
79   let rec lemma matching_nth (order: list order_trading) (b_order : Seq.
80   seq order) (s_order : Seq.seq order)
81     requires { matching order b_order s_order }
82     ensures { forall k :int. 0 ≤ k < Len.length order →
83     matching_order (nth k order) b_order s_order }
84     variant { order }
85     =
86     match order with
87     | Nil → ()
88     | Cons _ l → matching_nth l b_order s_order
89   end

```

```

87 let rec lemma matching_same_price (order: list order_trading) (b_order
   : Seq.seq order) (s_order : Seq.seq order) (b_order' : Seq.seq order
   ) (s_order' : Seq.seq order)
88   requires { matching order b_order s_order }
89   requires { Seq.length b_order = Seq.length b_order' }
90   requires { Seq.length s_order = Seq.length s_order' }
91   requires {forall j:int. 0 ≤ j < Seq.length b_order → b_order'[j].
   price_order = b_order[j].price_order }
92   requires {forall j:int. 0 ≤ j < Seq.length s_order → s_order'[j].
   price_order = s_order[j].price_order }
93   ensures { matching order b_order' s_order' }
94   variant { order }
95   =
96   match order with
97   | Nil → ()
98   | Cons _ l →
99     matching_same_price l b_order s_order b_order' s_order'
100   end
101
102
103 predicate smallest_buyer_seller (order: list order_trading) (buyer :
   int) (seller : int) =
104   match order with
105   | Nil → true
106   | Cons k l → smallest_buyer_seller l buyer seller ∧
107     k.buyer_index ≥ buyer ∧
108     k.seller_index ≥ seller
109   end
110
111
112 function sum_seller (l : list order_trading) (sellerIndexe : int) :
   int
113 =
114 match l with
115 | Nil → 0
116 | Cons h t → ( if h.seller_index = sellerIndexe then Uint.to_int(h.
   amount_t) else 0 ) + sum_seller t sellerIndexe
117 end
118
119 let rec lemma sum_seller_positive (l : list order_trading) (
   buyerIndexe : int)
120   ensures { 0 ≤ sum_seller l buyerIndexe }
121   =
122   match l with
123   | Nil → ()
124   | Cons _ l → sum_seller_positive (l : list order_trading) (
   buyerIndexe : int)
125   end
126
127 function sum_buyer (l : list order_trading) (buyerIndexe : int) : int

```



```

128 =
129 match l with
130 | Nil → 0
131 | Cons h t → ( if h.buyer_index = buyerIndexe then Uint.to_int(h.
amount_t) else 0 ) + sum_buyer t buyerIndexe end
132
133
134 let rec lemma sum_buyer_positive (l : list order_trading) (
buyerIndexe : int)
135 ensures { 0 ≤ sum_buyer l buyerIndexe }
136 =
137 match l with
138 | Nil → ()
139 | Cons _ l → sum_buyer_positive (l : list order_trading) (
buyerIndexe : int)
140 end
141
142 let rec lemma smallest_buyer_seller_sum_seller (order: list
order_trading) (buyer : int) (seller : int) (b_order : Seq.seq order
) (s_order : Seq.seq order)
143 requires { matching order b_order s_order }
144 requires { smallest_buyer_seller order buyer seller }
145 requires { sum_seller order seller = 0 }
146 ensures { smallest_buyer_seller order buyer (seller + 1) }
147 =
148 match order with
149 | Nil → ()
150 | Cons _ l →
151 smallest_buyer_seller_sum_seller (l: list order_trading) (buyer
: int) (seller : int) b_order s_order
152 end
153
154 let rec lemma smallest_buyer_seller_sum_buyer (order: list
order_trading) (buyer : int) (seller : int) (b_order : Seq.seq order
) (s_order : Seq.seq order)
155 requires { matching order b_order s_order }
156 requires { smallest_buyer_seller order buyer seller }
157 requires { sum_buyer order buyer = 0 }
158 ensures { smallest_buyer_seller order (buyer + 1) seller }
159 =
160 match order with
161 | Nil → ()
162 | Cons _ l →
163 smallest_buyer_seller_sum_buyer (l: list order_trading) (buyer
: int) (seller : int) b_order s_order
164 end
165
166 let rec lemma smallest_buyer_seller_expensive_seller (order: list
order_trading) (buyer : int) (seller : int) (b_order : Seq.seq order
) (s_order : Seq.seq order)

```

```

167     requires { matching order b_order s_order }
168     requires { sorted_order b_order }
169     requires { 0 ≤ buyer < Seq.length b_order }
170     requires { smallest_buyer_seller order buyer seller }
171     requires { b_order[buyer].price_order < s_order[seller].
price_order }
172     ensures { smallest_buyer_seller order buyer (seller + 1) }
173     variant { order }
174     =
175     match order with
176     | Nil → ()
177     | Cons _ l →
178         smallest_buyer_seller_expensive_seller (l: list order_trading
) (buyer : int) (seller : int) b_order s_order
179     end
180
181 let lemma smallest_buyer_seller_after_last (order: list order_trading
) (buyer : int) (seller : int) (b_order : Seq.seq order) (s_order :
Seq.seq order)
182     requires { matching order b_order s_order }
183     requires { smallest_buyer_seller order buyer seller }
184     requires { Seq.length s_order ≤ seller ∨ Seq.length b_order ≤
buyer }
185     ensures { order = Nil }
186     =
187     match order with
188     | Nil → ()
189     | Cons _ _ →
190         absurd
191     end
192
193
194 function nb_token (l : list order_trading) : int
195 =
196 match l with
197 | Nil → 0
198 | Cons h t → h.amount_t + nb_token t
199 end
200
201 let rec lemma nb_token_positive (l : list order_trading)
202     ensures { 0 ≤ nb_token l }
203 =
204 match l with
205 | Nil → ()
206 | Cons _ l → nb_token_positive (l : list order_trading)
207 end
208
209 let rec lemma nb_token_zero_sum_buyer (l : list order_trading) (
indexe : uint)
210     requires { nb_token l = 0 }

```

```

211   ensures { sum_seller l indexe = 0 }
212   ensures { sum_buyer l indexe = 0 }
213   =
214   match l with
215   | Nil → ()
216   | Cons _ l → nb_token_zero_sum_buyer (l : list order_trading) (
indexe : uint)
217   end
218
219 predicate correct (l:list order_trading) (buy_order: Seq.seq order) (
sell_order: Seq.seq order) =
220   (forall i:uint. 0 ≤ i < Seq.length sell_order →
221     sum_seller l i ≤ Uint.to_int(sell_order[i].tokens)) ∧
222   (forall i:uint. 0 ≤ i < Seq.length buy_order →
223     sum_buyer l i ≤ Uint.to_int(buy_order[i].tokens)) ∧
224   matching l buy_order sell_order
225
226 let rec ghost find_seller (l:list order_trading) (buy_order: Seq.seq
order) (sell_order: Seq.seq order) (buyer:uint) (seller:uint) : (list
order_trading , order_trading)
227   requires { matching l buy_order sell_order }
228   requires { smallest_buyer_seller l buyer seller }
229   requires { 0 < sum_seller l seller }
230   ensures { let l',_ = result in nb_token l = 1 + nb_token l' }
231   ensures { let l',k = result in
232     forall buyer. sum_buyer l buyer = sum_buyer l' buyer
+ (if k.buyer_index = buyer then 1 else 0) }
233   ensures { let l',k = result in
234     forall seller. sum_seller l seller = sum_seller l'
seller + (if k.seller_index = seller then 1 else 0) }
235   ensures { let l',_ = result in matching l' buy_order sell_order }
236   ensures { let l',_ = result in smallest_buyer_seller l' buyer
seller }
237   ensures { let _,k = result in k.seller_index = seller }
238   ensures { let _,k = result in k.buyer_index ≥ buyer }
239   ensures { let _,k = result in matching_order k buy_order
sell_order }
240   variant { l }
241   =
242   match l with
243   | Nil → absurd
244   | Cons k l →
245     if k.seller_index = seller then
246       if k.amount_t = 1 then l,k else (Cons {k with amount_t = k.
amount_t - 1} l), {k with amount_t = 1}
247     else
248       let l,k' = find_seller l buy_order sell_order buyer seller in
249       (Cons k l),k'
250   end
251

```

```

252 let rec ghost find_buyer (l:list order_trading) (buy_order: Seq.seq
    order) (sell_order: Seq.seq order) (buyer:uint) (seller:uint) : (list
    order_trading , order_trading)
253   requires { matching l buy_order sell_order }
254   requires { smallest_buyer_seller l buyer seller }
255   requires { 0 < sum_buyer l buyer }
256   ensures { let l',_ = result in nb_token l = 1 + nb_token l' }
257   ensures { let l',k = result in
258     forall buyer. sum_buyer l buyer = sum_buyer l' buyer
    + (if k.buyer_index = buyer then 1 else 0) }
259   ensures { let l',k = result in
260     forall seller. sum_seller l seller = sum_seller l'
    seller + (if k.seller_index = seller then 1 else 0) }
261   ensures { let l',_ = result in matching l' buy_order sell_order }
262   ensures { let l',_ = result in smallest_buyer_seller l' buyer
    seller }
263   ensures { let _,k = result in k.buyer_index = buyer }
264   ensures { let _,k = result in k.seller_index ≥ seller }
265   ensures { let _,k = result in matching_order k buy_order
    sell_order }
266   variant { 1 }
267   =
268   match l with
269   | Nil → absurd
270   | Cons k l →
271     if k.buyer_index = buyer then
272       if k.amount_t = 1 then l,k else (Cons {k with amount_t = k.
    amount_t - 1} l), {k with amount_t = 1}
273     else
274       let l,k' = find_buyer l buy_order sell_order buyer seller in
275       (Cons k l),k'
276   end
277
278 let ghost remove_seller_buyer_token1 (l:list order_trading) (
    buy_order: Seq.seq order) (sell_order: Seq.seq order) (buyer:uint) (
    seller:uint) : list order_trading
279   requires { sorted_order buy_order }
280   requires { sorted_order sell_order }
281   requires { matching l buy_order sell_order }
282   requires { smallest_buyer_seller l buyer seller }
283   requires { 1 ≤ sum_seller l seller }
284   requires { 1 ≤ sum_buyer l buyer }
285   requires { buy_order[buyer].price_order ≥ sell_order[seller].
    price_order }
286   ensures { nb_token l = 1 + nb_token result }
287   ensures { forall buyer'. sum_buyer l buyer' = sum_buyer result
    buyer' + (if buyer' = buyer then 1 else 0) }
288   ensures { forall seller'. sum_seller l seller' = sum_seller
    result seller' + (if seller' = seller then 1 else 0) }
289   ensures { matching result buy_order sell_order }

```

```

290     ensures { smallest_buyer_seller result buyer seller }
291   =
292     let l, k = find_seller l buy_order sell_order buyer seller in
293     if k.buyer_index = buyer then l
294     else
295       let l, k' = find_buyer l buy_order sell_order buyer seller in
296       assert { buy_order[k.buyer_index].price_order ≥ sell_order[seller
297         ].price_order };
298       assert { buy_order[buyer].price_order ≥ sell_order[k'.
299         seller_index].price_order };
300       Cons { buyer_index = k.buyer_index; seller_index = k'.seller_index
301         ; amount_t = 1 } l
302
303   let ghost remove_seller_token1 (l:list order_trading) (buy_order: Seq.
304     seq order) (sell_order: Seq.seq order) (buyer:uint) (seller:uint) :
305     list order_trading
306   requires { sorted_order buy_order }
307   requires { sorted_order sell_order }
308   requires { matching l buy_order sell_order }
309   requires { smallest_buyer_seller l buyer seller }
310   requires { 1 ≤ sum_seller l seller }
311   requires { buy_order[buyer].price_order ≥ sell_order[seller].
312     price_order }
313   ensures { nb_token l = 1 + nb_token result }
314   ensures { forall buyer'. sum_buyer l buyer' ≥ sum_buyer result
315     buyer' }
316   ensures { forall seller'. sum_seller l seller' = sum_seller
317     result seller' + (if seller' = seller then 1 else 0) }
318   ensures { matching result buy_order sell_order }
319   ensures { smallest_buyer_seller result buyer seller }
320   =
321     let l, _ = find_seller l buy_order sell_order buyer seller in
322     l
323
324   let ghost remove_buyer_token1 (l:list order_trading) (buy_order: Seq.
325     seq order) (sell_order: Seq.seq order) (buyer:uint) (seller:uint) :
326     list order_trading
327   requires { sorted_order buy_order }
328   requires { sorted_order sell_order }
329   requires { matching l buy_order sell_order }
330   requires { smallest_buyer_seller l buyer seller }
331   requires { 1 ≤ sum_buyer l buyer }
332   requires { buy_order[buyer].price_order ≥ sell_order[seller].
333     price_order }
334   ensures { nb_token l = 1 + nb_token result }
335   ensures { forall buyer'. sum_buyer l buyer' = sum_buyer result
336     buyer' + (if buyer' = buyer then 1 else 0) }
337   ensures { forall seller'. sum_seller l seller' ≥ sum_seller
338     result seller' }
339   ensures { matching result buy_order sell_order }

```

```

327     ensures { smallest_buyer_seller result buyer seller }
328   =
329     let l,_ = find_buyer l buy_order sell_order buyer seller in
330     l
331
332
333   let rec ghost remove_token1 (l:list order_trading) (buy_order: Seq.seq
order) (sell_order: Seq.seq order) (buyer:uint) (seller:uint) : list
order_trading
334     requires { sorted_order buy_order }
335     requires { sorted_order sell_order }
336     requires { matching l buy_order sell_order }
337     requires { smallest_buyer_seller l buyer seller }
338     requires { buy_order[buyer].price_order ≥ sell_order[seller].
price_order }
339     requires { 0 < nb_token l }
340     ensures { nb_token l = 1 + nb_token result }
341     ensures { forall buyer'. sum_buyer l buyer' ≥ sum_buyer result
buyer' }
342     ensures { forall seller'. sum_seller l seller' ≥ sum_seller
result seller' }
343     ensures { matching result buy_order sell_order }
344     ensures { smallest_buyer_seller result buyer seller }
345     variant { l }
346   =
347     match l with
348     | Nil → absurd
349     | Cons k l →
350       if k.amount_t = 1 then l
351       else Cons { k with amount_t = k.amount_t - 1 } l
352     end
353
354
355   let rec ghost remove_seller_buyer' (l:list order_trading) (buy_order:
Seq.seq order) (sell_order: Seq.seq order) (buyer:uint) (seller:uint)
) (token: uint) : list order_trading
356     requires { sorted_order buy_order }
357     requires { sorted_order sell_order }
358     requires { matching l buy_order sell_order }
359     requires { smallest_buyer_seller l buyer seller }
360     requires { buy_order[buyer].price_order ≥ sell_order[seller].
price_order }
361     ensures { nb_token l ≤ token + nb_token result }
362     ensures { forall buyer'. buyer' ≠ buyer → sum_buyer l buyer' ≥
sum_buyer result buyer' }
363     ensures { forall seller'. seller' ≠ seller → sum_seller l
seller' ≥ sum_seller result seller' }
364     ensures { max (sum_buyer l buyer - token) 0 = sum_buyer result
buyer }

```

```

365     ensures { max (sum_seller l seller - token) 0 = sum_seller result
seller }
366     ensures { matching result buy_order sell_order }
367     ensures { smallest_buyer_seller result buyer seller }
368     variant { token }
369     writes { }
370     reads { }
371   =
372     if token = 0 then l
373   else
374     let l =
375       if 0 < sum_seller l (Uint.to_int seller) && 0 < sum_buyer l (
Uint.to_int buyer)
376     then remove_seller_buyer_token1 l buy_order sell_order buyer
seller
377       else if 0 < sum_seller l (Uint.to_int seller) then
remove_seller_token1 l buy_order sell_order buyer seller
378       else if 0 < sum_buyer l (Uint.to_int buyer) then
remove_buyer_token1 l buy_order sell_order buyer seller
379       else if 0 < nb_token l then
remove_token1 l buy_order sell_order buyer seller
380       else
381         Nil
382     in
383     remove_seller_buyer' l buy_order sell_order buyer seller (token
-1)
384
385   (* Trading algorithm that matches sales and purchases *)
386   (* as input I have an array of buy orders and an array of sell orders
*)
387
388   let trading (buy_order : array order) (sell_order : array order) :
list order_trading
389   requires { Arr.length buy_order > 0 ∧ Arr.length sell_order > 0}
390   requires {sorted_order buy_order}
391   requires {sorted_order sell_order}
392   requires {forall j:int. 0 ≤ j < Arr.length buy_order → 0 <
buy_order[j].tokens }
393   requires {forall j:int. 0 ≤ j < Arr.length sell_order → 0 <
sell_order[j].tokens }
394   ensures { correct result (old buy_order) (old sell_order) }
395   ensures { forall l. correct l (old buy_order) (old sell_order) →
nb_token l ≤ nb_token result }
396   =
397   (*order_list the output of the function*)
398   (*order list that brings together the matching between seller and
buyer*)
399   let order_list : ref (list order_trading) = ref Nil in
400   let i = ref (0:uint) in
401   let j = ref (0:uint) in
402

```

```

406      (*I sort my arrays in a decreasing way*)
407      assert{sorted_order buy_order};
408      label Before in
409
410      let ghost others = ref (fun (l:list order_trading) → l) in
411      let ghost buy_order0 = pure { buy_order.elts } in
412      let ghost sell_order0 = pure { sell_order.elts } in
413
414      while Uint.(<) !i (Arr.length buy_order) && Uint.(<) !j (Arr.
length sell_order) do
415
416          invariant {0 ≤ !i ≤ Arr.length (buy_order at Before) ∧ 0 ≤ !j
≤ Arr.length (sell_order at Before)}
417          invariant {0 ≤ !i ≤ Arr.length (buy_order) ∧ 0 ≤ !j ≤ Arr.
length (sell_order )}
418          invariant {sorted_order (buy_order at Before)}
419          invariant {sorted_order (sell_order at Before)}
420
421          invariant {forall j:int. 0 ≤ j < Arr.length buy_order →
buy_order[j].orderAddress == (buy_order[j].orderAddress at Before)}
422          invariant {forall j:int. 0 ≤ j < Arr.length sell_order →
sell_order[j].orderAddress == (sell_order[j].orderAddress at Before)}
423
424          invariant {forall j:int. 0 ≤ j < Arr.length (buy_order at
Before) → (buy_order at Before)[j].price_order = buy_order[j].
price_order }
425          invariant {forall j:int. 0 ≤ j < Arr.length (sell_order at
Before) → (sell_order at Before)[j].price_order = sell_order[j].
price_order }
426
427          invariant {forall j:int. 0 ≤ j < Arr.length (buy_order at
Before) → Uint.to_int(buy_order[j].tokens) ≤ Uint.to_int((buy_order
at Before)[j].tokens) }
428          invariant {forall j:int. 0 ≤ j < Arr.length (sell_order at
Before) → Uint.to_int(sell_order[j].tokens) ≤ Uint.to_int((
sell_order at Before)[j].tokens) }
429
430          invariant {forall k:int. !i ≤ k < Arr.length (buy_order at
Before) → 0 < Uint.to_int(buy_order[k].tokens) }
431          invariant {forall k:int. !j ≤ k < Arr.length (sell_order at
Before) → 0 < Uint.to_int(sell_order[k].tokens) }
432
433          invariant {matching !order_list (buy_order at Before) (
sell_order at Before)}
434
435          invariant {forall i:uint. 0 ≤ i < Arr.length (sell_order at
Before) →
436                      sum_seller !order_list i + sell_order[i].
tokens = (sell_order at Before)[i].tokens }
437

```



```

448   invariant {forall i:uint. 0 ≤ i < Arr.length (buy_order at
Before) →
449       sum_buyer !order_list i + Uint.to_int(
buy_order[i].tokens) = Uint.to_int((buy_order at Before)[i].tokens)
    }
450   invariant { forall l. correct l (old buy_order) (old sell_order)
→
451       nb_token l ≤ nb_token !order_list +
nb_token (!others l) }
452   invariant { forall l. correct l (old buy_order) (old sell_order)
→
453       correct (!others l) buy_order sell_order }
454   invariant { forall l. correct l (old buy_order) (old sell_order)
→
455       smallest_buyer_seller (!others l) !i !j
    }
456   variant {Arr.length buy_order + Arr.length sell_order - !i - !j}
457   (*check if the purchase price offer is greater than or equal to
the selling price*)
458   if Uint.(≥) buy_order[!i].price_order sell_order[!j].
price_order then begin
459
460       (*check if the seller can provide me enough energy*)
461       if Uint.(≤) buy_order[!i].tokens sell_order[!j].tokens then
begin
462
463       (*if this is the case then the quantity transferred is
worth the requested quantity of the buyer*)
464       let amount_transferred = buy_order[!i].tokens in
465
466       let ghost others' = !others in
467       let ghost buyer = !i in
468       let ghost seller = !j in
469       let ghost buy_order' : Seq.seq order = buy_order.elts in
470       let ghost sell_order' : Seq.seq order = sell_order.elts in
471       others := (fun l → if pure { correct l buy_order0
sell_order0 }
472           then remove_seller_buyer' (others' l)
buy_order' sell_order' buyer seller amount_transferred
473           else l);
474
475       assert { forall l. correct l (old buy_order) (old
sell_order) →
476           matching (!others l) buy_order
sell_order
    };

```

```

472      (*I subtract from the seller the amount transferred, he can
      sell the energy he has in excess to another buyer*)
473      sell_order[!j] ← { sell_order[!j] with tokens = Uint.(-)
sell_order[!j].tokens buy_order[!i].tokens};
474      buy_order[!i] ← { buy_order[!i] with tokens = 0};
475      (*I have a seller a buyer and the transaction, I create a
      record*)
476      assert { forall k: int. 0 ≤ k < Arr.length sell_order → k
      ≠ !j → sell_order[k].orderAddress == (sell_order[k].orderAddress
      at Before) };
477      assert { forall k: int. 0 ≤ k < Arr.length buy_order → k
      ≠ !i → buy_order[k].orderAddress == (buy_order[k].orderAddress at
      Before) };
478
479      assert { forall l. correct l (old buy_order) (old
sell_order) →
480          matching (!others l) buy_order
sell_order      };
481      let registered_order = {
482          seller_index = !j;
483          buyer_index = !i;
484          amount_t = amount_transferred;
485      } in
486      assert { matching_order registered_order (buy_order at
      Before) (sell_order at Before) };
487
488      assert { forall j: int. 0 ≤ j < Arr.length sell_order →
      sell_order[j].orderAddress == (sell_order[j].orderAddress at Before)
      };
489
490      (*I add to my list the new matching*)
491      order_list := Cons registered_order !order_list;
492
493      assert { forall l. correct l (old buy_order) (old
sell_order) →
494          smallest_buyer_seller (!others l) !i !j
495          };
496
497      assert { forall l. correct l (old buy_order) (old
sell_order) →
498          sum_buyer (!others l) !i = 0 };
499      (*I go to the next buyer *)
500      i := !i + 1;
501      assert { forall l. correct l (old buy_order) (old
sell_order) →
502          smallest_buyer_seller (!others l) !i !j
503          };
504
505
506      (*

```

```

507         assert { forall l. correct l (old buy_order) (old
508         sell_order) →
509             nb_token l ≤ nb_token !order_list +
510             nb_token (!others l) };
511         assert { forall l. correct l (old buy_order) (old
512         sell_order) →
513             matching (!others l) buy_order sell_order
514             };
515         assert { forall l. correct l (old buy_order) (old
516         sell_order) →
517             forall k : int. 0 ≤ k < Len.length (!
518             others l) →
519                 !i ≤ (nth k (!others l)).buyer_index ∧
520                 !j ≤ (nth k (!others l)).seller_index
521             };
522 *)
523 (* if the seller has sold all of his energy, then I go to
524 the next seller *)
525 if sell_order[!j].tokens = 0 then begin
526     assert { forall l. correct l (old buy_order) (old
527     sell_order) →
528         sum_seller (!others l) !j = 0 };
529     j := !j+1;
530 end
531 (*if the seller does not have enough energy that the buyer wants
532 *)
533 end else begin
534     (*the amount of energy sent is worth the totality of energy
535     of the seller*)
536     let amount_transferred = sell_order[!j].tokens in
537     let ghost others' = !others in
538     let ghost buyer = !i in
539     let ghost seller = !j in
540     let ghost buy_order' : Seq.seq order = buy_order.elts in
541     let ghost sell_order' : Seq.seq order = sell_order.elts in
542     others := (fun l → if pure { correct l buy_order0
543     sell_order0 }
544         then remove_seller_buyer' (others' l)
545         buy_order' sell_order' buyer seller amount_transferred
546         else l);
547     (*I subtract from the buyer the amount of energy of the
548     seller, and what remains he can buy from another seller*)
549     buy_order[!i] ← { buy_order[!i] with tokens = Uint.(-)
550     buy_order[!i].tokens sell_order[!j].tokens};
551     sell_order[!j] ← { sell_order[!j] with tokens = 0 };
552     assert { forall k: int. 0 ≤ k < Arr.length sell_order → k
553     ≠ !j → sell_order[k].orderAddress == (sell_order[k].orderAddress at
554     Before) };

```

```

541     assert { forall k: int. 0 ≤ k < Arr.length buy_order → k
    ≠ !i → buy_order[k].orderAddress == (buy_order[k].orderAddress at
    Before) };
542     (*I create a new record that I will store in my order list*)
543     let registered_order = {
544         seller_index = !j;
545         buyer_index = !i;
546         amount_t = amount_transferred;
547     } in
548     order_list := Cons registered_order !order_list;
549     (*I go to the next seller so that the buyer can exchange
    with another seller*)
550     j := !j + 1
551     end
552 end
553 else begin
554     assert { forall l. correct l (old buy_order) (old sell_order)
    →
555         forall k :int. 0 ≤ k < Len.length (!others l) →
556             !j = (nth k (!others l)).seller_index →
557             sell_order[!j].price_order ≤ buy_order[(nth k (!
    others l)).buyer_index].price_order
558         };
559     assert { sorted_order buy_order };
560     j := !j + 1; (*in case there is no matching I go to the next
    seller*)
561     end
562     done;
563
564     (*I return my order list created*)
565     !order_list
566 end
567
568
569 module Gas
570     use int.Int
571     use ref.Ref
572     use bool.Bool
573
574     exception Out_of_gas
575     (*note that the add_gas function is different from that of the paper
    *)
576     (*Indeed, in this version we do ⊢ take into account the allocation
    parameter*)
577     (*the compilation and calculation of the number of gas consumed does
    ⊢ yet work*)
578     (*on our case study, but it is in progress. So we have simplify the
    add_gas function.*)
579     type gas = int
580     val ghost tot_gas : ref gas

```

```

581
582   val ghost add_gas (used : gas) : unit
583       requires { 0 ≤ used }
584       ensures { !tot_gas = (old !tot_gas) + used }
585       writes { tot_gas }
586
587 end
588
589 module ETPMarket
590   use my_library.Address
591   use my_library.UInt256
592   use my_library.Uint
593   use my_library.SmartMeterID
594   use mach.peano.Peano as Peano
595   (* use my_library.PeanoUInt160 as PeanoInt160 *)
596   use Gas
597   use int.Int
598   use ref.Ref
599   use Trading
600
601   type purchase = {amount_p: uint; price_p : uint} (*it can be buy ou
602       sell -- amount it's the energy in tokens*)
603
604   val marketOpen : ref bool
605   constant sell_gas_consumed : gas
606   constant buy_gas_consumed : gas
607
608   axiom sell_consumed: sell_gas_consumed ≥ 0
609   axiom buy_consumed: buy_gas_consumed ≥ 0
610
611   clone my_library.Hashtbl as Ord with
612       type key = Peano.t
613
614   type ord = {
615       mutable nextID: Peano.t;
616       ord: Ord.t order;
617   }
618   invariant { 0 ≤ nextID }
619   invariant { forall x:Peano.t. 0 ≤ x < nextID → Ord.mem_ ord x }
620   invariant { forall x:Peano.t. nextID ≤ x → ¬ (Ord.mem_ ord x) }
621   by {
622       nextID = Peano.zero;
623       ord = Ord.create ();
624   }
625
626   val sellOrd : ord
627   val buyOrd : ord
628
629   exception WhenMarketOpen (*modifier WhenMarketOpen*)

```

```

630  (* cf https://gitlab.inria.fr/why3/why3/merge\_requests/201 *)
631  axiom injectivity: forall x y: Peano.t. (x:int) = y → x = y
632
633  (*private function *)
634  let eTPMarket_sell (_sell_purch : purchase) : unit
635    requires { !marketOpen }
636    requires { (_sell_purch.amount_p) > 0 }
637    requires { (_sell_purch.price_p) > 0 }
638
639    (*the function add a new order*)
640    ensures { (Ord.sizee sell0rd.ord) = (Ord.sizee (old sell0rd.ord)
+ 1) }
641
642    (*I found in the hashtable the sell order I recorded*)
643    ensures {let order = Ord.find_ sell0rd.ord (old sell0rd.nextID)
in
644      order.tokens = _sell_purch.amount_p ∧
645      order.price_order = _sell_purch.price_p ∧
646      order.orderAddress = msg_sender
647    }
648
649    ensures {!tot_gas - old !tot_gas ≤ sell_gas_consumed}
650  =
651    let sell_order = {
652      orderAddress = msg_sender; (*msg sender is the
account address that calls this function, the seller*)
653      tokens = _sell_purch.amount_p;
654      price_order = _sell_purch.price_p;
655    } in
656
657    Ord.add sell0rd.ord sell0rd.nextID sell_order;
658    sell0rd.nextID ← Peano.succ sell0rd.nextID;
659    add_gas (sell_gas_consumed)
660
661  (*private function*)
662  let eTPMarket_buy (_buy_purch : purchase) : unit
663    requires { !marketOpen }
664    requires { _buy_purch.amount_p > 0 }
665    requires { _buy_purch.price_p > 0 }
666    ensures { (Ord.sizee buy0rd.ord) = (Ord.sizee (old buy0rd.ord) +
1) }
667
668    ensures {let order = Ord.find_ buy0rd.ord (old buy0rd.nextID) in
order.orderAddress = msg_sender ∧
669      order.tokens = _buy_purch.amount_p ∧
670      order.price_order = _buy_purch.price_p
671    }
672
673    ensures {!tot_gas - old !tot_gas ≤ buy_gas_consumed}
674  =

```

```

675     let buy_order = {orderAddress = msg_sender; (*msg sender is the
676     potential buyer who will call the buy function*)
677     tokens = _buy_purch.amount_p;
678     price_order = _buy_purch.price_p;} in
679     Ord.add buyOrd.ord buyOrd.nextID buy_order;
680     buyOrd.nextID ← Peano.succ buyOrd.nextID; (*the mapping stores
681     any purchase *)
682     add_gas (buy_gas_consumed)
683 end
684 module ETPMarketBisBis
685   use int.Int
686   use ref.Ref
687   use bool.Bool
688   use my_library.Address
689   use my_library.Uint
690   use ETPMarket
691   use Gas
692
693   val algorithm : ref address
694   val onlyOwner : ref bool
695   val owner : address
696
697   constant open_gas_consumed : gas
698   constant close_gas_consumed : gas
699   constant setAlgo_gas_consumed : gas
700
701   axiom open_gas: open_gas_consumed ≥ 0
702   axiom close_gas: close_gas_consumed ≥ 0
703   axiom setAlgo_gas: setAlgo_gas_consumed ≥ 0
704
705   exception OnlyOwner
706   exception MarketOpen
707   exception MarketClose
708
709   (* public function *)
710   let openMarket () : unit
711     ensures {!tot_gas - old !tot_gas ≤ open_gas_consumed}
712     raises {MarketOpen → !marketOpen = True}
713   =
714     if !marketOpen then raise MarketOpen;
715     marketOpen := True;
716     add_gas (open_gas_consumed)
717
718   (* public function *)
719   let closeMarket () : unit
720     ensures {!tot_gas - old !tot_gas ≤ close_gas_consumed}
721     raises {MarketClose → !marketOpen = False}

```

```

723     =
724     if ¬ !marketOpen then raise MarketClose;
725     marketOpen := False;
726     sellOrd.nextID ← Peano.zero;
727     Ord.clear sellOrd.ord;
728     buyOrd.nextID ← Peano.zero;
729     Ord.clear buyOrd.ord;
730     add_gas (close_gas_consumed)
731
732     (* public function *)
733     let eTPMarket_setAlgorithm (_algorithmAddress : address)
734         raises {OnlyOwner → !onlyOwner = False}
735     =
736     if ¬ (!onlyOwner) then raise OnlyOwner;
737     algorithm := _algorithmAddress;
738     add_gas (setAlgo_gas_consumed)
739
740
741 end
742
743 module ETPAccount
744     use int.Int
745     use my_library.Address
746     use my_library.UInt256
747     use my_library.Uint
748     use Gas
749     use ETPMarket
750     use bool.Bool
751     use ref.Ref
752
753     constant asell_gas_consumed : gas
754     constant abuy_gas_consumed : gas
755     constant acomplete_gas_consumed : gas
756
757     axiom asell_gas: asell_gas_consumed ≥ 0
758     axiom abuy_gas: abuy_gas_consumed ≥ 0
759     axiom acomplete_gas: acomplete_gas_consumed ≥ 0
760
761     (*private function*)
762     let eTPAccount_sell (_sell_pursh : purchase)
763         requires { !marketOpen}
764         requires {(_sell_pursh.amount_p) > 0}
765         requires {(_sell_pursh.price_p) > 0}
766     =
767         eTPMarket_sell (_sell_pursh);
768         add_gas (asell_gas_consumed)
769
770
771     (* private function *)
772     let eTPAccount_buy (_buy_pursh : purchase)

```



```

773     requires { !marketOpen}
774     requires {(_buy_pursh.amount_p) > 0}
775     requires {(_buy_pursh.price_p) > 0}
776     =
777       eTPMarket_buy (_buy_pursh);
778       add_gas (abuy_gas_consumed)
779
780
781
782     (* private function *)
783     let eTPAccount_complete (_sellerAddress : address) (_callerFunction
: address) (_price : uint) : unit
784     requires {acceptableEtherTransaction balance _callerFunction
_sellerAddress ( _price)}
785     requires {uniqueAddress _sellerAddress _callerFunction }
786     requires {(_price) > 0}
787     ensures {etherTransactionCompletedSuccessfully (old balance)
balance _sellerAddress _callerFunction}
788     =
789       address_send (UInt256.v_of_uint (_price)) _callerFunction
_sellerAddress;
790       add_gas (acomplete_gas_consumed)
791 end
792
793 module ETPRegistryBis
794   use my_library.UInt256
795   use my_library.SmartMeterID
796   use my_library.Address
797   use my_library.Uint
798   use Gas
799   use ETPMarketBisBis
800   use ETPAccount
801   use ETPMarket
802   use int.EuclideanDivision
803   use int.Power
804   use int.Int
805   use ref.Ref
806   use bool.Bool
807   use Trading
808   use DCC
809
810   val market : ref address
811   val oracle : address
812   val defAddress : address
813   val onlyOracle : ref bool
814
815   let constant floatingPointCorrection : uint = 0x10000000
816   constant setMarket_gas_consumed : gas
817   constant register_gas_consumed : gas
818   constant record_gas_consumed : gas

```

```

819
820 axiom setMarket_gas: setMarket_gas_consumed  $\geq$  0
821 axiom register_gas: register_gas_consumed  $\geq$  0
822 axiom record_gas: record_gas_consumed  $\geq$  0
823
824 clone my_library.Hashtbl as AddressOf with
825     type key = smartMeterID
826
827 val exportBalanceOf : Bal.t uint
828 val importBalanceOf : Bal.t uint
829 val marketBalanceOf : Bal.t uint
830 val addressOf : AddressOf.t address
831
832 exception OnlyOracle (*modifier OnlyOracle*)
833 exception OwnerNotFound
834 exception ExistingSmartMeter
835 exception NoSmartMeter
836 exception NoAmount
837 exception OverFlow
838 exception ExistingRecord
839 exception ExistingOrder
840 exception ZeroNumber
841 exception MarketNotFound
842 exception ExistingMarket
843 exception NoPrice
844
845 (* public function *)
846 let eTPRegistry_setMarket (_market : address)
847     raises {OnlyOwner  $\rightarrow$  !onlyOwner = False}
848     raises {ExistingMarket  $\rightarrow$  !market = _market}
849 =
850     if  $\neg$  !onlyOwner then raise OnlyOwner;
851     if (!market == _market) then raise ExistingMarket;
852     market := _market;
853     add_gas (setMarket_gas_consumed)
854
855 (* public function *)
856 let registerSmartMeter (_meterID : smartMeterID) (_ownerAddress :
address)
857     raises { OnlyOwner $\rightarrow$  !onlyOwner = False }
858     raises {ExistingSmartMeter  $\rightarrow$  AddressOf.mem_ addressOf _meterID}
859     ensures { (AddressOf.sizee addressOf) = (AddressOf.sizee (old
addressOf) + 1 ) }
860     ensures { AddressOf.mem_ addressOf _meterID}
861 =
862     if  $\neg$  (!onlyOwner) then raise OnlyOwner;
863     if AddressOf.mem addressOf _meterID then raise ExistingSmartMeter
;
864     AddressOf.add addressOf _meterID _ownerAddress;
865     add_gas (register_gas_consumed)

```

```

866
867   (* public function *)
868   let recordImportsAndExports (pot_buy : pot_buy) (pot_sell :
pot_sell)
869       raises {OnlyOracle → !onlyOracle = False }
870       raises {NoSmartMeter → ¬ AddressOf.mem_ addressOf pot_buy.smb_id
      ∨ ¬ AddressOf.mem_ addressOf pot_sell.sms_id}
871       raises {OwnerNotFound → AddressOf.([]) addressOf pot_buy.smb_id
= defAddress ∨ AddressOf.([]) addressOf pot_sell.sms_id =
defAddress}
872       raises {WhenMarketOpen → ¬ !marketOpen}
873       raises {NoAmount → pot_sell.amount_s = zero_unsigned ∨ pot_buy
.amount_b = zero_unsigned}
874       raises {Overflow → (pot_sell.amount_s) > div (max_uint) ((
floatingPointCorrection)) ∨
875         (pot_buy.amount_b) > div (max_uint) ((
floatingPointCorrection)) ∨
876         (pot_sell.amount_s) * (floatingPointCorrection) >
max_uint ∨
877         (pot_buy.amount_b) * (floatingPointCorrection) >
max_uint }
878       raises {ExistingRecord → Bal.mem_ exportBalanceOf (AddressOf
.([]) addressOf pot_sell.sms_id)
879         ∨ Bal.mem_ importBalanceOf (AddressOf.([]) addressOf
pot_buy.smb_id) }
880       raises {ZeroNumber → floatingPointCorrection = zero_unsigned}
881       raises {ExistingMarket → Bal.mem_ marketBalanceOf !market}
882       raises {NoPrice → pot_sell.price_s ≤ 0 ∨ pot_buy.price_b ≤ 0}
883   =
884     if ¬ !marketOpen then raise WhenMarketOpen;
885     if ¬ (!onlyOracle) then raise OnlyOracle;
886     if ¬ AddressOf.mem addressOf pot_buy.smb_id then raise
NoSmartMeter;
887     if ¬ AddressOf.mem addressOf pot_sell.sms_id then raise
NoSmartMeter;
888
889     let owner_s = AddressOf.find_def addressOf pot_sell.sms_id
defAddress in
890     if owner_s == defAddress then raise OwnerNotFound;
891
892     let owner_b = AddressOf.find_def addressOf pot_buy.smb_id
defAddress in
893     if owner_b == defAddress then raise OwnerNotFound;
894     if pot_buy.amount_b = 0 then raise NoAmount;
895     if pot_sell.amount_s = 0 then raise NoAmount;
896     if floatingPointCorrection = 0 then raise ZeroNumber;
897     if (pot_sell.amount_s) > (Uint.(/) (Uint.of_int(max_uint))
floatingPointCorrection) then raise Overflow;
898     if (pot_buy.amount_b) > (Uint.(/) (Uint.of_int(max_uint))
floatingPointCorrection) then raise Overflow;

```

```

899     let exportWithCorrection = (pot_sell.amount_s) * (
floatingPointCorrection) in
900     if Bal.mem exportBalanceOf owner_s then raise ExistingRecord;
901     if Bal.mem importBalanceOf owner_b then raise ExistingRecord;
902     if pot_sell.price_s ≤ 0 then raise NoPrice;
903     if pot_buy.price_b ≤ 0 then raise NoPrice;
904
905     let export_purchase = {
906         amount_p = exportWithCorrection;
907         price_p = pot_sell.price_s;
908     } in
909     Bal.add exportBalanceOf owner_s ((export_purchase).amount_p);
910
911     let importWithCorrection = (pot_buy.amount_b) * (
floatingPointCorrection) in
912     let import_purchase = {
913         amount_p = importWithCorrection;
914         price_p = pot_buy.price_b;
915     } in
916
917     Bal.add importBalanceOf owner_b ((import_purchase).amount_p);
918
919     if Bal.mem marketBalanceOf !market then raise ExistingMarket;
920     Bal.add marketBalanceOf !market 0;
921     if (pot_buy.amount_b > 0) then eTPAccount_buy(import_purchase)
922     else eTPAccount_sell(export_purchase);
923     add_gas (record_gas_consumed)
924
925 end
926
927 module ETPRegistry
928     use int.Int
929     use my_library.UInt256
930     use my_library.SmartMeterID
931     use my_library.Address
932     use my_library.Uint
933     use ref.Ref
934     use ETPMarket
935     use Gas
936     use ETPRegistryBis
937     use bool.Bool
938
939
940     val onlymarket : ref bool (*modifier*)
941     constant transferTo_gas_consumed : gas
942     constant transferFrom_gas_consumed : gas
943
944     axiom transferTo_gas: transferTo_gas_consumed ≥ 0
945     axiom transferFrom_gas: transferFrom_gas_consumed ≥ 0
946

```

```

947  (* private function *)
948  let transferToMarket (_from : address) (_value : uint) : unit (*
    value are green tokens to send *)
949
950      requires {!onlymarket}
951      requires { _value > 0 }
952      requires { (Bal.([]) marketBalanceOf !market) = 0 }
953      requires { acceptableAmountTransaction exportBalanceOf
marketBalanceOf _from !market _value}
954      ensures {amountTransactionCompletedSuccessfully (old
exportBalanceOf) exportBalanceOf (old marketBalanceOf)
marketBalanceOf _from !market }
955  =
956      amount_transaction (exportBalanceOf) (marketBalanceOf) (_from) (!
market) (_value);
957      add_gas (transferTo_gas_consumed)
958
959  (* private function *)
960  let transferFromMarket (_to : address) (_value : uint) : unit (*
    _value = green token*)
961
962      requires {!onlymarket}
963      requires { _value > 0 }
964      requires {(Bal.([]) marketBalanceOf !market) > 0}
965      requires {acceptableAmountTransaction marketBalanceOf
importBalanceOf !market _to _value}
966      ensures {amountTransactionCompletedSuccessfully (old
marketBalanceOf) marketBalanceOf (old importBalanceOf)
importBalanceOf !market _to}
967
968  =
969      amount_transaction (marketBalanceOf) (importBalanceOf) (!market) (
_to) (_value);
970      add_gas (transferFrom_gas_consumed)
971
972  end
973
974  module ETPMarketBis
975      use int.Int
976      use my_library.SmartMeterID
977      use my_library.Address
978      use my_library.UInt256
979      use my_library.Uint
980      use Gas
981      use ETPMarket
982      use ETPAccount
983      use ETPRegistry
984      use ETPRegistryBis
985      use ref.Ref
986      use Trading

```

```

987
988
989
990
991 val onlyAlgo : ref bool (*modifier*)
992 constant mcomplete_gas_consumed : gas
993
994 axiom mcomplete_gas: mcomplete_gas_consumed ≥ 0
995
996 (* private function *)
997 let eTPMarket_complete (sellId: Peano.t) (buyId : Peano.t) (
998   _purchase : purchase) : unit
999   requires {!onlymarket}
1000   requires { (_purchase.amount_p) > 0 ∧ (_purchase.price_p) > 0 }
1001   requires {(Bal.([]) marketBalanceOf !market) > 0}
1002   requires {acceptableAmountTransaction marketBalanceOf
importBalanceOf !market ((Ord.([]) buyOrd.ord buyId).orderAddress)
_purchase.amount_p}
1003   requires {acceptableEtherTransaction balance (Ord.([]) buyOrd.ord
buyId).orderAddress (Ord.([]) sellOrd.ord sellId).orderAddress (
_purchase.price_p)}
1004   requires {!onlyAlgo}
1005   requires { sellId ≥ 0 ∧ buyId ≥ 0 }
1006   requires {Ord.mem_ sellOrd.ord sellId}
1007   requires {Ord.mem_ buyOrd.ord buyId}
1008
1009   requires {uniqueAddress (Ord.([]) sellOrd.ord sellId).
orderAddress (Ord.([]) buyOrd.ord buyId).orderAddress}
1010
1011
1012   ensures {etherTransactionCompletedSuccessfully (old balance)
balance (Ord.([]) buyOrd.ord buyId).orderAddress (Ord.([]) sellOrd.
ord sellId).orderAddress}
1013   ensures {amountTransactionCompletedSuccessfully (old
importBalanceOf) importBalanceOf (old marketBalanceOf)
marketBalanceOf (Ord.([]) buyOrd.ord buyId).orderAddress !market}
1014   =
1015   let sellOrder = Ord.([]) sellOrd.ord sellId in
1016   let buyOrder = Ord.([]) buyOrd.ord buyId in
1017   eTPAccount_complete (sellOrder.orderAddress) (buyOrder.
orderAddress) (_purchase.price_p);
1018   transferFromMarket (buyOrder.orderAddress) (_purchase.amount_p);
1019   add_gas (mcomplete_gas_consumed)
1020 end

```

Appendix B : WCET of function with allocation

```

1 type list α = Nil | Cons α (list α)
2

```

```

3 function length (l: list  $\alpha$ ) : int =
4   match l with
5   | Nil      → 0
6   | Cons _ r → 1 + length r
7   end
8
9 let rec length_ [@ evm:gas_checking] (l:list  $\alpha$ ) : int32
10 requires { (length l) ≤ max_int32 }
11 ensures { !gas - old !gas ≤ (length l) * 128 + 71 }
12 ensures { !alloc - old !alloc ≤ 0 }
13 ensures { result = length l }
14 variant { l } =
15   match l with
16   | Nil → add_gas 71 0; 0
17   | Cons _ l → add_gas 128 0; 1 + length_ l
18   end
19
20 let rec mk_list42 [@ evm:gas_checking] (i:int32) : list int32
21 requires { 0 ≤ i }
22 ensures { !gas - old !gas ≤ i * 185 + 113 }
23 ensures { !alloc - old !alloc ≤ i * 96 + 32 }
24 ensures { i = length result }
25 variant { i } =
26   if i ≤ 0 then (add_gas 113 32; Nil) else
27     let l = mk_list42 (i-1) in
28       add_gas 185 96;
29       Cons (0x42:int32) l
30
31 let g_ [@ evm:gas_checking] (i:int32) : int32
32 requires { 0 ≤ i }
33 ensures { !gas - old !gas ≤ i * 313 + 242 }
34 ensures { !alloc - old !alloc ≤ i * 96 + 32 } =
35   add_gas 58 0;
36   let l = mk_list42 i in
37   length_ l

```