Deductive Proof of Industrial Smart Contracts
Using Why3

Zeinab Nehai¹,² 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 \( \sim \$60M \). 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 library 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].

### 2.2 Why3 features intended for Smart Contracts

**Library modelling.** Solidity is an imperative object-oriented programming language, characterized by static typing[4]. 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 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFF... >` 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 an `orderAddress` which can be a seller or a buyer, `tokens` that express the crypto-Kilowatt-hours (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).

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.

```solidity
function transferFromMarket (address _to, uint _value) onlyMarket returns (bool success) {
    if (exportBalanceOf[market] >= _value) {
        /* Transferring _value from market to _to */
    } else { success = false;
      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). 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:

```why3
let transferFromMarket (_to : address) (_value : uint) : bool 
requires {onlymarket ∧ _value > 0 }
requires {marketBalanceOf[market] ≥ _value }
requires {importBalanceOf[_to] ≤ max_uint - _value}
ensures { (old marketBalanceOf[market]) + (old importBalanceOf[_to]) = marketBalanceOf[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:

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.

```
function registerSmartMeter(string _meterId, address _ownerAddress) onlyOwner
{
    addressOf[_meterId] = _ownerAddress;
    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:

```
let registerSmartMeter (meterID : string) (ownerAddress : address)
    raises { OnlyOwner → !onlyOwner = False }
    raises {ExistingSmartMeter → mem addressOf meterID}
    ensures { (size addressOf) = (size (old addressOf) + 1 ) }
    ensures { mem addressOf meterID} = (*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 \texttt{buy\_order} and \texttt{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:

\begin{verbatim}
let trading (buy_order : array order) (sell_order : array order) : list order_trading
  requires { length buy_order > 0 ∧ length sell_order > 0 }
  requires { sorted_order buy_order }
  requires {forall j:int. 0 ≤ j < length buy_order → 0 < buy_order[j].tokens }
  requires {forall j:int. 0 ≤ j < length sell_order → 0 < sell_order[j].tokens } 
  ensures { forall l. correct l (old buy_order) (old sell_order) → 
    nb_token l ≤ nb_token result }
  ensures {!gas ≤ old !gas + 374 + (length buy_order + length sell_order) * 363}
  ensures {!alloc ≤ old !alloc + 35 + (length buy_order + length sell_order) * 35}

= (* The program *)
\end{verbatim}

- RTE properties: \textit{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: \textit{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: \textit{(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: \texttt{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.

\textit{(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).

\textit{(3) Gas consumption}; Lines 9 and 10 ensures that no extra-consumption of gas will happen (see the following paragraph).

\textit{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 $W_{verylow} = \{ADD, SUB, \ldots\}$, the amount
to pay is $G_{verylow} = 3$ units of gas, and for a create operation the amount
to pay is $G_{create} = 32000$ 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) 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 compilation 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 [] 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 datatypes 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 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.

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 expansive 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.
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. Incs, 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., Zanele-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., Olicel, 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

module DCC (*the module that materializes the smart meters*)
use my_library.Uint
use my_library.SmartMeterID
use my_library.Address
use array.Array

(*records of potential selleur and buyeur, with the purchase (price_b) and sale (price_s) price*)
(*amount_b the needed token quantity, and amount_s the token quantity on sale*)

type pot_buy = {address_b : address;
  smb_id: smartMeterID;
  price_b: uint;
  amount_b: uint}

type pot_sell = {address_s : address;
  sms_id : smartMeterID;
  price_s: uint;
  amount_s: uint}

(*buy_array and sell_array are data tables retrieved from the meters*)
val buy_array : array pot_buy
val sell_array : array pot_sell

module Trading
use my_library.Uint
use int.Int
use int.MinMax
use seq.Seq
use import my_library.ArrayUint as Arr
use ref.Refint
use list.List
use import list.Length as Len
use list.NthNoOpt
use my_library.SmartMeterID
use my_library.Address
use list.HdTlNoOpt
use list.Nth as Elem
type order = {orderAddress : address; tokens: uint; price_order: uint}

(*It can be buy or sell, tokens = energy materializes in token*)

close array.Sort as Sort with type elt = order

val sorted_array (a: array order) : unit
ensures {forall i j: int. 0 ≤ j ≤ i < Arr.length a → Uint.to_int(a[i].price_order) ≤ Uint.to_int(a[j].price_order)}
writes {a}

predicate sorted_order (a: Seq.seq order) =
forall k1 k2 : int. 0 ≤ k1 ≤ k2 < Seq.length a →
Uint.to_int(a[k2].price_order) ≤ Uint.to_int(a[k1].price_order)

(* *)

type order_trading = {seller_index: uint; buyer_index: uint; amount_t: uint}

predicate matching_order (k: order_trading) (b_order : Seq.seq order) (s_order : Seq.seq order) =
s_order[k.seller_index].price_order ≤
b_order[k.buyer_index].price_order ∧
0 ≤ k.buyer_index < Seq.length b_order ∧
0 ≤ k.seller_index < Seq.length s_order ∧
0 ≤ k.amount_t

predicate matching (order: list order_trading) (b_order : Seq.seq order) (s_order : Seq.seq order) =
match order with
| Nil → true
| Cons k l → matching l b_order s_order ∧
  matching_order k b_order s_order
end

let rec lemma matching_nth (order: list order_trading) (b_order : Seq.seq order) (s_order : Seq.seq order)
requires { matching order b_order s_order }
ensures {forall k :int. 0 ≤ k < Len.length order →
  matching_order (nth k order) b_order s_order }
variant { order } =
match order with
| Nil → ()
| Cons _ l → matching_nth l b_order s_order
end
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)
  requires { matching order b_order s_order }
  requires { Seq.length b_order = Seq.length b_order' }
  requires { Seq.length s_order = Seq.length s_order' }
  requires {forall j:int. 0 ≤ j < Seq.length b_order → b_order'[j].price_order = b_order[j].price_order }
  requires {forall j:int. 0 ≤ j < Seq.length s_order → s_order'[j].price_order = s_order[j].price_order }
  ensures { matching order b_order' s_order' }
  variant { order } =
match order with
| Nil → ()
| Cons _ l →
  matching_same_price l b_order s_order b_order' s_order'
end

predicate smallest_buyer_seller (order: list order_trading) (buyer : int) (seller : int) =
match order with
| Nil → true
| Cons k l → smallest_buyer_seller l buyer seller ∧
  k.buyer_index ≥ buyer ∧
  k.seller_index ≥ seller
end

function sum_seller (l : list order_trading) (sellerIndexe : int) : int =
match l with
| Nil → 0
| Cons h t → ( if h.seller_index = sellerIndexe then Uint.to_int(h.amount_t) else 0 ) + sum_seller t sellerIndexe
end

let rec lemma sum_seller_positive (l : list order_trading) (buyerIndexe : int)
  ensures { 0 ≤ sum_seller l buyerIndexe }
  =
match l with
| Nil → ()
| Cons _ l → sum_seller_positive (l : list order_trading) (buyerIndexe : int)
end

function sum_buyer (l : list order_trading) (buyerIndexe : int) : int
match l with
| Nil → 0
| Cons h t → (if h.buyer_index = buyerIndexe then Uint.to_int(h.
amount_t) else 0) + sum_buyer t buyerIndexe end

let rec lemma sum_buyer_positive (l : list order_trading) (buyerIndexe : int)
ensures { 0 ≤ sum_buyer l buyerIndexe }
=
match l with
| Nil → ()
| Cons _ l → sum_buyer_positive (l : list order_trading) (buyerIndexe : int)
end

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)
requires { matching order b_order s_order }
requires { smallest_buyer_seller order buyer seller }
requires { sum_seller order seller = 0 }
ensures { smallest_buyer_seller order buyer (seller + 1) }
=
match order with
| Nil → ()
| Cons _ l → smallest_buyer_seller_sum_seller (l: list order_trading) (buyer : int) (seller : int) b_order s_order
end

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)
requires { matching order b_order s_order }
requires { smallest_buyer_seller order buyer seller }
requires { sum_buyer order buyer = 0 }
ensures { smallest_buyer_seller order (buyer + 1) seller }
=
match order with
| Nil → ()
| Cons _ l → smallest_buyer_seller_sum_buyer (l: list order_trading) (buyer : int) (seller : int) b_order s_order
end

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)
requires { matching order b_order s_order }
requires { sorted_order b_order }
requires { 0 ≤ buyer < Seq.length b_order }
requires { smallest_buyer_seller order buyer seller }
requires { b_order[buyer].price_order < s_order[seller].price_order }
ensures { smallest_buyer_seller order buyer (seller + 1) }
variant { order }
= 
match order with 
| Nil → ()
| Cons _ l → smallest_buyer_seller_expensive_seller (l: list order_trading) (buyer : int) (seller : int) b_order s_order end

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)
requires { matching order b_order s_order }
requires { smallest_buyer_seller order buyer seller }
requires { Seq.length s_order ≤ seller ∨ Seq.length b_order ≤ buyer }
ensures { order = Nil }
= 
match order with 
| Nil → ()
| Cons _ _ → absurd end

function nb_token (l : list order_trading) : int
= 
match l with 
| Nil → 0
| Cons h t → h.amount_t + nb_token t end

let rec lemma nb_token_positive (l : list order_trading)
ensures { 0 ≤ nb_token l}
= 
match l with 
| Nil → ()
| Cons _ l → nb_token_positive (l : list order_trading) end

let rec lemma nb_token_zero_sum_buyer (l : list order_trading) (indexe : uint)
requires { nb_token l = 0 }
ensures { \text{sum\_seller}\(l\) indexe = 0 } 

ensures { \text{sum\_buyer}\(l\) indexe = 0 } 

= 

match \(1\) with 
| \text{Nil} \rightarrow () 
| \text{Cons } _\text{1} \rightarrow \text{nb\_token\_zero\_sum\_buyer}(l : \text{list\ order trading})(\indexe : \text{uint}) 
end 

\text{predicate correct} \((l : \text{list\ order trading})(\text{buy\_order} : \text{Seq\_seq\ order})(\text{sell\_order} : \text{Seq\_seq\ order}) = \)
\(\text{forall } \text{i:uint}. \ 0 \leq i < \text{Seq\_length\ sell\_order} \rightarrow \)
\(\text{sum\_seller}\ l \ i \leq \text{Uint\_to\_int(sell\_order}[\text{i}].\text{tokens}) \land \)
\(\text{forall } \text{i:uint}. \ 0 \leq i < \text{Seq\_length\ buy\_order} \rightarrow \)
\(\text{sum\_buyer}\ l \ i \leq \text{Uint\_to\_int(buy\_order}[\text{i}].\text{tokens}) \land \)
\text{matching\ l\ buy\_order\ sell\_order} 

\text{let rec ghost find\_seller} \((l : \text{list\ order trading})(\text{buy\_order} : \text{Seq\_seq\ order})(\text{sell\_order} : \text{Seq\_seq\ order})(\text{buyer:uint})(\text{seller:uint}) : (\text{list\ order\ trading}, \text{order\ trading}) = \)
\text{requires} { \text{matching\ l\ buy\_order\ sell\_order} } 
\text{requires} { \text{smallest\_buyer\_seller\ l\ buyer\ seller} } 
\text{requires} { \ 0 < \text{sum\_seller}\ l\ \text{seller} } 
\text{ensures} { \text{let } l',_ = \text{result in } \text{nb\_token}\ l = 1 + \text{nb\_token}\ l' } 
\text{ensures} { \text{let } l',k = \text{result in } \text{forall}\ \text{buyer}\ . \ \text{sum\_buyer}\ l\ \text{buyer} = \text{sum\_buyer}\ l'\ \text{buyer} + (\text{if } \text{k\_buyer\_index} = \text{buyer} \text{ then } 1 \text{ else } 0) } 
\text{ensures} { \text{let } l',k = \text{result in } \text{forall}\ \text{seller}\ . \ \text{sum\_seller}\ l\ \text{seller} = \text{sum\_seller}\ l'\ \text{seller} + (\text{if } \text{k\_seller\_index} = \text{seller} \text{ then } 1 \text{ else } 0) } 
\text{ensures} { \text{let } l',_ = \text{result in } \text{matching}\ l'\ \text{buy\_order}\ \text{sell\_order} } 
\text{ensures} { \text{let } l',_ = \text{result in } \text{smallest\_buyer\_seller}\ l'\ \text{buyer\ seller} } 
\text{ensures} { \text{let } _,k = \text{result in } \text{k\_seller\_index} = \text{seller} } 
\text{ensures} { \text{let } _,k = \text{result in } \text{k\_buyer\_index} \geq \text{buyer} } 
\text{ensures} { \text{let } _,k = \text{result in } \text{matching\_order}\ k\ \text{buy\_order}\ \text{sell\_order} } 
\text{variant} { \ (1) } 

= 

match \(1\) with 
| \text{Nil} \rightarrow \text{absurd} 
| \text{Cons } _\text{k} \rightarrow \)
if \(\text{k\_seller\_index} = \text{seller} \text{ then } \)
if \(\text{k\_amount\_t} = 1 \text{ then } \text{l},\text{k} \text{ else } (\text{Cons } \{\text{k \text{ with } amount\_t} = \text{k}.\text{amount\_t} - 1\} \text{ l}), \{\text{k \text{ with } amount\_t} = 1\} \)
else 
let \(l',k' = \text{find\_seller}\ l\ \text{buy\_order}\ \text{sell\_order}\ \text{buyer}\ \text{seller} \text{ in } \)
(\text{Cons } k\ l),k' 
end
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)
  requires { matching l buy_order sell_order }
  requires { smallest_buyer_seller l buyer seller }
  requires { 0 < sum_buyer l buyer }
  ensures { let 1',_ = result in nb_token l = 1 + nb_token 1' }
  ensures { let 1',k = result in
    forall buyer. sum_buyer l buyer = sum_buyer 1' buyer + (if k.buyer_index = buyer then 1 else 0) }
  ensures { let 1',k = result in
    forall seller. sum_seller l seller = sum_seller 1' seller + (if k.seller_index = seller then 1 else 0) }
  ensures { let 1',_ = result in matching 1' buy_order sell_order }
  ensures { let 1',_ = result in smallest_buyer_seller 1' buyer seller }
  ensures { let _,k = result in k.buyer_index = buyer }
  ensures { let _,k = result in k.seller_index >= seller }
  ensures { let _,k = result in matching_order k buy_order sell_order }
  variant { l }
  =
  match l with
  | Nil → absurd
  | Cons k l →
    if k.buyer_index = buyer then
      if k.amount_t = 1 then l,k else (Cons {k with amount_t = k.amount_t - 1} l), {k with amount_t = 1}
    else
      let l,k' = find_buyer l buy_order sell_order buyer seller in
      (Cons k l),k'
end

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
  requires { sorted_order buy_order }
  requires { sorted_order sell_order }
  requires { matching l buy_order sell_order }
  requires { smallest_buyer_seller l buyer seller }
  requires { 1 <= sum_seller l seller }
  requires { 1 <= sum_buyer l buyer }
  requires { buy_order[buyer].price_order >= sell_order[seller].price_order }
  ensures { nb_token l = 1 + nb_token result }
  ensures { for all buyer'. sum_buyer l buyer' = sum_buyer result buyer' + (if buyer' = buyer then 1 else 0) }
  ensures { for all seller'. sum_seller l seller' = sum_seller result seller' + (if seller' = seller then 1 else 0) }
  ensures { matching result buy_order sell_order }
ensures { smallest_buyer_seller result buyer seller } =
let l, k = find_seller l buy_order sell_order buyer seller in
if k.buyer_index = buyer then l
else
  let l, k' = find_buyer l buy_order sell_order buyer seller in
  assert { buy_order[k.buyer_index].price_order \geq \ sell_order[seller].price_order };
  assert { buy_order[buyer].price_order \geq \ sell_order[k'.seller_index].price_order };
  Cons { buyer_index = k.buyer_index; seller_index = k'.seller_index ; amount_t = 1 } l

let ghost remove_seller_token1 (l:list order_trading) (buy_order: Seq.seq order) (sell_order: Seq.seq order) (buyer:uint) (seller:uint) : list order_trading
  requires { sorted_order buy_order } 
  requires { sorted_order sell_order } 
  requires { matching l buy_order sell_order } 
  requires { smallest_buyer_seller l buyer seller } 
  requires { 1 \leq \ text{sum_seller} l seller } 
  requires { buy_order[buyer].price_order \geq \ sell_order[seller].price_order } 
  ensures { nb_token l = 1 + nb_token result } 
  ensures { forall buyer'. text{sum_buyer} l buyer' \geq \ text{sum_buyer result buyer'} } 
  ensures { forall seller'. text{sum_seller} l seller' = text{sum_seller result seller'} + (if seller' = seller then 1 else 0) } 
  =
  let l, _ = find_seller l buy_order sell_order buyer seller in
  l

let ghost remove_buyer_token1 (l:list order_trading) (buy_order: Seq.seq order) (sell_order: Seq.seq order) (buyer:uint) (seller:uint) : list order_trading
  requires { sorted_order buy_order } 
  requires { sorted_order sell_order } 
  requires { matching l buy_order sell_order } 
  requires { smallest_buyer_seller l buyer seller } 
  requires { 1 \leq \ text{sum_buyer} l buyer } 
  requires { buy_order[buyer].price_order \geq \ sell_order[seller].price_order } 
  ensures { nb_token l = 1 + nb_token result } 
  ensures { forall buyer'. text{sum_buyer} l buyer' = text{sum_buyer result buyer'} + (if buyer' = buyer then 1 else 0) } 
  ensures { forall seller'. text{sum_seller} l seller' \geq \ text{sum_seller result seller'} } 
  ensures { matching result buy_order sell_order }
ensures { smallest_buyer_seller result buyer seller } =
  let l, _ = find_buyer l buy_order sell_order buyer seller in
  l

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
  requires { sorted_order buy_order }
  requires { sorted_order sell_order }
  requires { matching l buy_order sell_order }
  requires { smallest_buyer_seller l buyer seller }
  requires { buy_order[buyer].price_order \geq sell_order[seller].price_order }
  requires { 0 < nb_token l }
  ensures { nb_token l = 1 + nb_token result }
  ensures { forall buyer'. sum_buyer l buyer' \geq sum_buyer result buyer' }
  ensures { forall seller'. sum_seller l seller' \geq sum_seller result seller' }
  variant { l }
  =
  match l with
  | Nil -> absurd
  | Cons k l ->
  if k.amount_t = 1 then l
  else Cons { k with amount_t = k.amount_t - 1 } l
end

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
  requires { sorted_order buy_order }
  requires { sorted_order sell_order }
  requires { matching l buy_order sell_order }
  requires { smallest_buyer_seller l buyer seller }
  requires { buy_order[buyer].price_order \geq sell_order[seller].price_order }
  requires { nb_token l \leq token + nb_token result }
  requires { forall buyer'. buyer' \neq buyer \rightarrow sum_buyer l buyer' \geq sum_buyer result buyer' }
  requires { forall seller'. seller' \neq seller \rightarrow sum_seller l seller' \geq sum_seller result seller' }
  ensures { max (sum_buyer l buyer - token) 0 = sum_buyer result buyer }

ensures { max (sum_seller l seller - token) 0 = sum_seller result seller }
ensures { matching result buy_order sell_order }
ensures { smallest_buyer_seller result buyer seller }
variant { token }
writes { }
reads { }
=
if token = 0 then l
else
  let l =
    if 0 < sum_seller l (Uint.to_int seller) && 0 < sum_buyer l (Uint.to_int buyer)
    then remove_seller_buyer_token1 l buy_order sell_order buyer seller
    else if 0 < sum_seller l (Uint.to_int seller) then
      remove_seller_token1 l buy_order sell_order buyer seller
    else if 0 < sum_buyer l (Uint.to_int buyer) then
      remove_buyer_token1 l buy_order sell_order buyer seller
    else if 0 < nb_token l then
      remove_token1 l buy_order sell_order buyer seller
    else
      Nil
  in
  remove_seller_buyer' l buy_order sell_order buyer seller (token -1)

(* Trading algorithm that matches sales and purchases *)
(* as input I have an array of buy orders and an array of sell orders *)
let trading (buy_order : array order) (sell_order : array order) : list order_trading
requires { Arr.length buy_order > 0 ∧ Arr.length sell_order > 0}
requires {sorted_order buy_order}
requires {sorted_order sell_order}
requires {forall j:int. 0 ≤ j < Arr.length buy_order ∧ 0 < buy_order[j].tokens }
requires {forall j:int. 0 ≤ j < Arr.length sell_order ∧ 0 < sell_order[j].tokens }
ensures { correct result (old buy_order) (old sell_order) }
ensures { forall l. correct 1 (old buy_order) (old sell_order) ⇒ nb_token l ≤ nb_token result }
=
(*order_list the output of the function*)
(*order list that brings together the matching between seller and buyer*)
let order_list : ref (list order_trading) = ref Nil in
let i = ref (0:uint) in
let j = ref (0:uint) in
(*I sort my arrays in a decreasing way*)
assert{sorted_order buy_order};
label Before in

let ghost others = ref (fun (l:list order_trading) → l) in
let ghost buy_order0 = pure { buy_order.elts } in
let ghost sell_order0 = pure { sell_order.elts } in

while Uint.(<) !i (Arr.length buy_order) && Uint.(<) !j (Arr.
length sell_order) do

  invariant {0 ≤ !i ≤ Arr.length (buy_order at Before) ∧ 0 ≤ !j
             ≤ Arr.length (sell_order at Before)}
  invariant {0 ≤ !i ≤ Arr.length (buy_order) ∧ 0 ≤ !j ≤ Arr.
            length (sell_order )}
  invariant {sorted_order (buy_order at Before)}
  invariant {sorted_order (sell_order at Before)}

  invariant {forall j: int. 0 ≤ j < Arr.length buy_order →
            buy_order[j].orderAddress == (buy_order[j].orderAddress at Before)}
  invariant {forall j: int. 0 ≤ j < Arr.length sell_order →
            sell_order[j].orderAddress == (sell_order[j].orderAddress at Before)}

  invariant {forall j:int. 0 ≤ j < Arr.length (buy_order at
                Before) → (buy_order at Before)[j].price_order = buy_order[j].
                price_order }
  invariant {forall j:int. 0 ≤ j < Arr.length (sell_order at
                Before) → (sell_order at Before)[j].price_order = sell_order[j].
                price_order }

  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) }
  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) }

  invariant {forall k:int. !i ≤ k < Arr.length (buy_order at
                Before) → 0 < (Uint.to_int(buy_order[k].tokens) }
  invariant {forall k:int. !j ≤ k < Arr.length (sell_order at
                Before) → 0 < (Uint.to_int(sell_order[k].tokens) }

  invariant {matching !order_list (buy_order at Before) (sell_order at Before)}

  invariant {forall i:uint. 0 ≤ i < Arr.length (sell_order at
                Before) →
             sum_seller !order_list i + sell_order[i].
             tokens = (sell_order at Before)[i].tokens }

  invariant {forall k:int. !i ≤ k < Arr.length (buy_order at
                Before) → 0 < (Uint.to_int(buy_order[k].tokens) }
  invariant {forall k:int. !j ≤ k < Arr.length (sell_order at
                Before) → 0 < (Uint.to_int(sell_order[k].tokens) }

  invariant {matching !order_list (buy_order at Before) (sell_order at Before)}
invariant {\( \forall i: \text{uint}. \ 0 \leq i < \text{Arr.length} \ (\text{buy\_order at Before}) \rightarrow \sum_{\text{buyer}} \text{order\_list} i + \text{Uint.to_int} (\text{buy\_order}[i].\text{tokens}) = \text{Uint.to_int} ((\text{buy\_order at Before})[i].\text{tokens}) \)}

\begin{align*}
\text{invariant } & \{ \forall l. \text{correct} \ l \ (\text{old buy\_order}) \ (\text{old sell\_order}) \\
& \rightarrow \text{nb\_token} \ l \leq \text{nb\_token} \ (\text{\text{order\_list}}) \} \\
\text{invariant } & \{ \forall l. \text{correct} \ l \ (\text{old buy\_order}) \ (\text{old sell\_order}) \\
& \rightarrow \text{correct} \ (\text{\text{others} l}) \text{buy\_order sell\_order} \} \\
\text{invariant } & \{ \forall l. \text{correct} \ l \ (\text{old buy\_order}) \ (\text{old sell\_order}) \\
& \rightarrow \text{smallest\_buyer\_seller} \ (\text{\text{others} l}) !i !j \}
\end{align*}

\begin{align*}
\text{variant } & \{ \text{Arr.length} \text{buy\_order} + \text{Arr.length} \text{sell\_order} - !i - !j \} \\
& \{ \text{*check if the purchase price offer is greater than or equal to the selling price*} \} \\
& \text{if} \text{Uint.(\(\geq\)) buy\_order}[!i].\text{price\_order} \text{sell\_order}[!j].\text{price\_order} \text{then begin} \\
& \{ \text{*check if the seller can provide me enough energy*} \} \\
& \text{if} \text{Uint.(\(\leq\)) buy\_order}[!i].\text{tokens} \text{sell\_order}[!j].\text{tokens} \text{then begin} \\
& \{ \text{*if this is the case then the quantity transferred is worth the requested quantity of the buyer*} \} \\
& \text{let} \text{amount\_transferred} = \text{buy\_order}[!i].\text{tokens} \text{in} \\
& \text{let} \text{ghost} \ others' = \text{\text{others} in} \\
& \text{let} \text{ghost} \ \text{buyer} = !i \text{ in} \\
& \text{let} \text{ghost} \ \text{seller} = !j \text{ in} \\
& \text{let} \text{ghost} \ \text{buy\_order}' : \text{Seq.seq} \text{order} = \text{buy\_order}.\text{elts} \text{ in} \\
& \text{let} \text{ghost} \ \text{sell\_order}' : \text{Seq.seq} \text{order} = \text{sell\_order}.\text{elts} \text{ in} \\
& \text{others} := (\text{fun} \ l \rightarrow \text{if pure} \{ \text{correct} \ l \ \text{buy\_order0} \}) \text{sell\_order0} \}
\end{align*}

\begin{align*}
\text{then remove\_seller\_buyer'} \ (\text{others'} l) \\
\text{buy\_order'} \ \text{sell\_order'} \ \text{buyer} \ \text{seller} \ \text{amount\_transferred} \\
\text{else} \ 1); \\
\text{assert } \{ \forall l. \text{correct} \ l \ (\text{old buy\_order}) \ (\text{old sell\_order}) \rightarrow \\
\text{matching} \ (\text{\text{others} l}) \text{buy\_order} \}
\end{align*}
(*I subtract from the seller the amount transferred, he can sell the energy he has in excess to another buyer*)

\[
sell\_order[!j] \leftarrow \{\text{sell\_order}[!j] \text{ with } \text{tokens} = \text{Uint}(-)
n\}\text{tokens buy\_order}[!i].\text{tokens};
\]

\[
buy\_order[!i] \leftarrow \{\text{buy\_order}[!i] \text{ with } \text{tokens} = 0\};
\]

(*I have a seller a buyer and the transaction, I create a record*)

\[
\text{assert } \{\text{forall } k: \text{int. } 0 \leq k < \text{Arr.length sell\_order} \rightarrow k \neq !j \rightarrow \text{sell\_order}[k].\text{orderAddress} = (\text{sell\_order}[k].\text{orderAddress at Before})\};
\]

\[
\text{assert } \{\text{forall } k: \text{int. } 0 \leq k < \text{Arr.length buy\_order} \rightarrow k \neq !i \rightarrow \text{buy\_order}[k].\text{orderAddress} = (\text{buy\_order}[k].\text{orderAddress at Before})\};
\]

\[
\text{assert } \{\text{forall } l. \text{correct } l \text{ (old buy\_order) (old sell\_order) } \rightarrow \text{matching } (!\text{others } l) \text{ buy\_order sell\_order}\};
\]

\[
\text{let registered\_order} = \{\text{seller\_index} = !j; \text{buyer\_index} = !i; \text{amount\_t} = \text{amount\_transferred};\}\text{ in assert } \{\text{matching\_order registered\_order (buy\_order at Before) (sell\_order at Before)}\};
\]

\[
\text{assert } \{\text{forall } j: \text{int. } 0 \leq j < \text{Arr.length sell\_order} \rightarrow \text{sell\_order}[j].\text{orderAddress} = (\text{sell\_order}[j].\text{orderAddress at Before})\};
\]

(*I add to my list the new matching*)

\[
\text{order\_list} := \text{Cons registered\_order }!\text{order\_list};
\]

\[
\text{assert } \{\text{forall } l. \text{correct } l \text{ (old buy\_order) (old sell\_order) } \rightarrow \text{smallest\_buyer\_seller } (!\text{others } l) !i !j\};
\]

\[
\text{assert } \{\text{forall } l. \text{correct } l \text{ (old buy\_order) (old sell\_order) } \rightarrow \text{sum\_buyer } (!\text{others } l) !i = 0\};
\]

(*I go to the next buyer*)

\[
i := !i + 1;
\]

\[
\text{assert } \{\text{forall } l. \text{correct } l \text{ (old buy\_order) (old sell\_order) } \rightarrow \text{smallest\_buyer\_seller } (!\text{others } l) !i !j\};
\]
assert { forall l. correct l (old buy_order) (old sell_order) \(\rightarrow\) nb_token l \(\leq\) nb_token !order_list + nb_token (!others l) };
assert { forall l. correct l (old buy_order) (old sell_order) \(\rightarrow\) matching (!others l) buy_order sell_order };
assert { forall l. correct l (old buy_order) (old sell_order) \(\rightarrow\) forall k : int. 0 \(\leq\) k < Len.length (!others l) \(\rightarrow\) !i \(\leq\) (nth k (!others l)).buyer_index \(\wedge\) !j \(\leq\) (nth k (!others l)).seller_index };

(* if the seller has sold all of his energy, then I go to the next seller *)
if sell_order[!j].tokens = 0 then begin
assert { forall l. correct l (old buy_order) (old sell_order) \(\rightarrow\) sum_seller (!others l) !j = 0 };
j := !j+1;
end

(*if the seller does not have enough energy that the buyer wants *)
end else begin
(*the amount of energy sent is worth the totality of energy of the seller*)
let amount_transfered = sell_order[!j].tokens in

let ghost others’ = !others in
let ghost buyer = !i in
let ghost seller = !j in
let ghost buy_order’ : Seq.seq order = buy_order.elts in
let ghost sell_order’ : Seq.seq order = sell_order.elts in
others := (fun l \rightarrow if pure { correct l buy_order0 sell_order0 } then remove_seller_buyer’ (others’ l) buy_order’ sell_order’ buyer seller amount_transfered else l);

(*I subtract from the buyer the amount of energy of the seller, and what remains he can buy from another seller*)
buy_order[!i] \leftarrow\{ buy_order[!i] with tokens = Uint.(-) buy_order[!i].tokens sell_order[!j].tokens \};
sell_order[!j] \leftarrow\{ sell_order[!j] with tokens = 0 \};
assert { forall k: int. 0 \(\leq\) k < Arr.length sell_order \(\rightarrow\) k \(\neq\) !j \(\rightarrow\) sell_order[k].orderAddress == (sell_order[k].orderAddress at Before) };
assert { forall k: int. 0 ≤ k < Arr.length buy_order → k ≠ !i → buy_order[k].orderAddress == (buy_order[k].orderAddress at Before) };

(*I create a new record that I will store in my order list*)
let registered_order = {
    seller_index = !j;
    buyer_index = !i;
    amount_t = amount_transferred;
} in
order_list := Cons registered_order !order_list;
(*I go to the next seller so that the buyer can exchange with another seller*)
j := !j + 1
end
else begin
    assert { forall l. correct l (old buy_order) (old sell_order) →
        forall k :int. 0 ≤ k < Len.length (!others l) →
        !j = (nth k (!others l)).seller_index →
        sell_order[!j].price_order ≤ buy_order[(nth k (! others l)).buyer_index].price_order
    };
    assert { sorted_order buy_order };
    j := !j + 1; (*in case there is no matching I go to the next seller*)
    end
end
done;

(*I return my order list created*)
!order_list

module Gas
use int.Int
use ref.Ref
use bool.Bool

exception Out_of_gas

(*note that the add_gas function is different from that of the paper *)
(*Indeed, in this version we do not take into account the allocation parameter*)
(*the compilation and calculation of the number of gas consumed does not yet work*)
(*on our case study, but it is in progress. So we have simplify the add_gas function.*)

type gas = int
val ghost tot_gas : ref gas
val ghost add_gas (used : gas) : unit
  requires { 0 ≤ used }
  ensures { !tot_gas = (old !tot_gas) + used }
  writes { tot_gas }
end

module ETPMarket
use my_library.Address
use my_library.UInt256
use my_library.Uint
use my_library.SmartMeterID
use mach.peano.Peano as Peano
(* use my_library.PeanoUint160 as PeanoInt160 *)
use Gas
use int.Int
use ref.Ref
use Trading

type purchase = {amount_p: uint; price_p : uint} (*it can be buy ou
sell -- amount it's the energy in tokens*)

val marketOpen : ref bool
constant sell_gas_consumed : gas
constant buy_gas_consumed : gas

axiom sell_consumed: sell_gas_consumed ≥ 0
axiom buy_consumed: buy_gas_consumed ≥ 0

clone my_library.Hashtbl as Ord with
  type key = Peano.t

  type ord = {
    mutable nextID: Peano.t;
    ord: Ord.t order;
  }
  invariant { 0 ≤ nextID }
  invariant { forall x:Peano.t. 0 ≤ x < nextID → Ord.mem ord x }
  invariant { forall x:Peano.t. nextID ≤ x → ¬(Ord.mem ord x) }
  by {
    nextID = Peano.zero;
    ord = Ord.create ();
  }

val sellOrd : ord
val buyOrd : ord

exception WhenMarketOpen (*modifier WhenMarketOpen*)
axiom injectivity: forall x y: Peano.t (x:int) = y → x = y

(*private function*)
let eTPMarket_sell (_sell_purch : purchase) : unit
  requires { !marketOpen }
  requires {(_sell_purch.amount_p) > 0 }
  requires {(_sell_purch.price_p) > 0 }
  (*the function add a new orders*)
  ensures { (Ord.sizee sellOrd.ord) = (Ord.sizee (old sellOrd.ord) + 1) }
  (*I found in the hashtable the sell order I recorded*)
  ensures {let order = Ord.find_ sellOrd.ord (old sellOrd.nextID) in
    order.tokens = _sell_purch.amount_p ∧
    order.price_order = _sell_purch.price_p ∧
    order.orderAddress = msg_sender }
  ensures {!tot_gas - old !tot_gas ≤ sell_gas_consumed}
= let sell_order = {
  orderAddress = msg_sender; (*msg sender is the account address that calls this function, the seller*)
  tokens = _sell_purch.amount_p;
  price_order = _sell_purch.price_p;
} in
Ord.add sellOrd.ord sellOrd.nextID sell_order;
sellOrd.nextID ← Peano.succ sellOrd.nextID;
add_gas (sell_gas_consumed)

(*private function*)
let eTPMarket_buy (_buy_purch : purchase) : unit
  requires { !marketOpen }
  requires { _buy_purch.amount_p > 0 }
  requires { _buy_purch.price_p > 0 }
  ensures { (Ord.sizee buyOrd.ord) = (Ord.sizee (old buyOrd.ord) + 1) }
  ensures {let order = Ord.find_ buyOrd.ord (old buyOrd.nextID) in
    order.orderAddress = msg_sender ∧
    order.tokens = _buy_purch.amount_p ∧
    order.price_order = _buy_purch.price_p }
  ensures {!tot_gas - old !tot_gas ≤ buy_gas_consumed}
=
let buy_order = {orderAddress = msg_sender; (* msg sender is the potential buyer who will call the buy function *)
    tokens = _buy_purch.amount_p;
    price_order = _buy_purch.price_p; } in
Ord.add buyOrd.ord buyOrd.nextID buy_order;
buyOrd.nextID ← Peano.succ buyOrd.nextID; (* the mapping stores any purchase *)
add_gas (buy_gas_consumed)

end

module ETPMarketBisBis
use int.Int
use ref.Ref
use bool.Bool
use my_library.Address
use my_library.Uint
use ETPMarket
use Gas

val algorithm : ref address
val onlyOwner : ref bool
val owner : address
constant open_gas_consumed : gas
constant close_gas_consumed : gas
constant setAlgo_gas_consumed : gas

axiom open_gas: open_gas_consumed ≥ 0
axiom close_gas: close_gas_consumed ≥ 0
axiom setAlgo_gas: setAlgo_gas_consumed ≥ 0

exception OnlyOwner
exception MarketOpen
exception MarketClose

(* public function *)
let openMarket () : unit
  ensures {!tot_gas - old !tot_gas ≤ open_gas_consumed}
  raises {MarketOpen → !marketOpen = True}
  =
    if !marketOpen then raise MarketOpen;
    marketOpen := True;
    add_gas (open_gas_consumed)

(* public function *)
let closeMarket () : unit
  ensures {!tot_gas - old !tot_gas ≤ close_gas_consumed}
  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  end
741
742  module ETPAccount
743  use int.Int
744  use my_library.Address
745  use my_library.UInt256
746  use my_library.Uint
747  use Gas
748  use ETPMarket
749  use bool.Bool
750  use ref.Ref
751
752  constant asell_gas_consumed : gas
753  constant abuy_gas_consumed : gas
754  constant acomplete_gas_consumed : gas
755
756  axiom asell_gas: asell_gas_consumed ≥ 0
757  axiom abuy_gas: abuy_gas_consumed ≥ 0
758  axiom acomplete_gas: acomplete_gas_consumed ≥ 0
759
760  (*private function*)
761  let eTPAccount_sell (_sell_pursh : purchase)
762     requires { !marketOpen}
763     requires {(_sell_pursh.amount_p) > 0}
764     requires {(_sell_pursh.price_p) > 0}
765     =
766     eTPMarket_sell (_sell_pursh);
767     add_gas (asell_gas_consumed)
768
769
770  (* private function *)
771  let eTPAccount_buy (_buy_pursh : purchase)
requires {!marketOpen}
requires {(_buy_pursh.amount_p) > 0}
requires {(_buy_pursh.price_p) > 0}

= eTPMarket_buy (_buy_pursh);
add_gas (abuy_gas_consumed)

(* private function *)
let eTPAccount_complete (_sellerAddress : address) (_callerFunction : address) (_price : uint) : unit
requires {acceptableEtherTransaction balance _callerFunction _sellerAddress (_price)}
requires {uniqueAddress _sellerAddress _callerFunction}
requires {(_price) > 0}
ensures {etherTransactionCompletedSuccessfully (old balance) balance _sellerAddress _callerFunction}
= address_send (UInt256.v_of_uint (_price)) _callerFunction _sellerAddress;
add_gas (acomplete_gas_consumed)
end

module ETPRegistryBis
use my_library.UInt256
use my_library.SmartMeterID
use my_library.Address
use my_library.Uint
use Gas
use ETPMarketBisBis
use ETPAccount
use ETPMarket
use int.EuclideanDivision
use int.Power
use int.Int
use ref.Ref
use bool.Bool
use Trading
use DCC
val market : ref address
val oracle : address
val defAddress : address
val onlyOracle : ref bool

let constant floatingPointCorrection : uint = 0x10000000
constant setMarket_gas_consumed : gas
constant register_gas_consumed : gas
constant record_gas_consumed : gas
axiom setMarket_gas: setMarket_gas_consumed ≥ 0
axiom register_gas: register_gas_consumed ≥ 0
axiom record_gas: record_gas_consumed ≥ 0

clone my_library.Hashtbl as AddressOf with
  type key = smartMeterID

val exportBalanceOf : Bal.t uint
val importBalanceOf : Bal.t uint
val marketBalanceOf : Bal.t uint
val addressOf : AddressOf.t address

exception OnlyOracle (*modifier OnlyOracle*)
exception OwnerNotFound
exception ExistingSmartMeter
exception NoSmartMeter
exception NoAmount
exception OverFlow
exception ExistingRecord
exception ExistingOrder
exception ZeroNumber
exception MarketNotFound
exception ExistingMarket
exception NoPrice

(* public function *)
let eTPRegistry_setMarket (_market : address)
  raises {OnlyOwner → !onlyOwner = False}
  raises {ExistingMarket → !market = _market}
  =
    if ¬ !onlyOwner then raise OnlyOwner;
    if (!market == _market) then raise ExistingMarket;
    market := _market;
    add_gas (setMarket_gas_consumed)

(* public function *)
let registerSmartMeter (_meterID : smartMeterID) (_ownerAddress : address)
  raises { OnlyOwner→ !onlyOwner = False } 
  raises {ExistingSmartMeter → AddressOf.mem_ addressOf _meterID}
  ensures { (AddressOf.sizee addressOf) = (AddressOf.sizee (old addressOf) + 1 ) }
  =
    if ¬ (!onlyOwner) then raise OnlyOwner;
    if AddressOf.mem addressOf _meterID then raise ExistingSmartMeter
      ;
      AddressOf.add addressOf _meterID _ownerAddress;
    add_gas (register_gas_consumed)
(* public function *)

let recordImportsAndExports (pot_buy : pot_buy) (pot_sell : pot_sell)
  raises {OnlyOracle → !onlyOracle = False }
  raises {NoSmartMeter → ¬ AddressOf.mem_ addressOf pot_buy.smb_id
            ∨ ¬ AddressOf.mem_ addressOf pot_sell.sms_id}
  raises {OwnerNotFound → AddressOf.([ ]) addressOf pot_buy.smb_id
          = defAddress ∨ AddressOf.([ ]) addressOf pot_sell.sms_id = defAddress}
  raises {WhenMarketOpen → ¬ !marketOpen}
  raises {NoAmount → pot_sell.amount_s = zero_unsigned ∨ pot_buy .amount_b = zero_unsigned}
  raises {OverFlow → (pot_sell.amount_s) > div (max_uint) ((
        floatingPointCorrection)) ∨
    (pot_buy.amount_b) > div (max_uint) ((
        floatingPointCorrection)) ∨
    (pot_sell.amount_s) * (floatingPointCorrection) >
    max_uint ∨
    (pot_buy.amount_b) * (floatingPointCorrection) >
    max_uint }
  raises {ExistingRecord → Bal.mem_ exportBalanceOf (AddressOf .([ ]) addressOf pot_sell.sms_id)
            ∨ Bal.mem_ importBalanceOf (AddressOf.([ ]) addressOf pot_buy.smb_id)}
  raises {ZeroNumber → floatingPointCorrection = zero_unsigned}
  raises {ExistingMarket → Bal.mem_ marketBalanceOf !market}
  raises {NoPrice → pot_sell.price_s ≤ 0 ∨ pot_buy.price_b ≤ 0}
    =
    if ¬ !marketOpen then raise WhenMarketOpen;
    if ¬ (!onlyOracle) then raise OnlyOracle;
    if ¬ AddressOf.mem addressOf pot_buy.smb_id then raise
          NoSmartMeter;
    if ¬ AddressOf.mem addressOf pot_sell.sms_id then raise
          NoSmartMeter;
    let owner_s = AddressOf.find_def addressOf pot_sell.sms_id
          defAddress in
    if owner_s == defAddress then raise OwnerNotFound;
    let owner_b = AddressOf.find_def addressOf pot_buy.smb_id
          defAddress in
    if owner_b == defAddress then raise OwnerNotFound;
    if pot_buy.amount_b = 0 then raise NoAmount;
    if pot_sell.amount_s = 0 then raise NoAmount;
    if floatingPointCorrection = 0 then raise ZeroNumber;
    if (pot_sell.amount_s) > (Uint.(/) (Uint.of_int(max_uint))
        floatingPointCorrection) then raise OverFlow;
    if (pot_buy.amount_b) > (Uint.(/) (Uint.of_int(max_uint))
        floatingPointCorrection) then raise OverFlow;
let exportWithCorrection = (pot_sell.amount_s) * (floatingPointCorrection) in
  if Bal.mem exportBalanceOf owner_s then raise ExistingRecord;
  if Bal.mem importBalanceOf owner_b then raise ExistingRecord;
  if pot_sell.price_s <= 0 then raise NoPrice;
  if pot_buy.price_b <= 0 then raise NoPrice;

let export_purchase = {
  amount_p = exportWithCorrection;
  price_p = pot_sell.price_s;
} in
  Bal.add exportBalanceOf owner_s ((export_purchase).amount_p);

let importWithCorrection = (pot_buy.amount_b) * (floatingPointCorrection) in
  let import_purchase = {
    amount_p = importWithCorrection;
    price_p = pot_buy.price_b;
  } in
    Bal.add importBalanceOf owner_b ((import_purchase).amount_p);

if Bal.mem marketBalanceOf !market then raise ExistingMarket;
  Bal.add marketBalanceOf !market 0;
if (pot_buy.amount_b > 0) then eTPAccount_buy(import_purchase)
else eTPAccount_sell(export_purchase);
add_gas (record_gas_consumed)
(* private function *)

let transferToMarket (_from : address) (_value : uint) : unit (*
value are green tokens to send *)

  requires {!onlymarket}
  requires { _value > 0 }
  requires { (Bal.([]) marketBalanceOf !market) = 0 }
  requires { acceptableAmountTransaction exportBalanceOf
    marketBalanceOf _from !market _value}
  ensures {amountTransactionCompletedSuccessfully (old
    exportBalanceOf) exportBalanceOf (old marketBalanceOf)
    marketBalanceOf _from !market }
  =
  amount_transaction (exportBalanceOf) (marketBalanceOf) (_from) (!
    market) (_value);
  add_gas (transferTo_gas_consumed)

(* private function *)

let transferFromMarket (_to : address) (_value : uint) : unit (*
    _value = green token*)

  requires {!onlymarket}
  requires { _value > 0 }
  requires { (Bal.([]) marketBalanceOf !market) > 0 }
  requires { acceptableAmountTransaction marketBalanceOf
    importBalanceOf !market _to _value}
  ensures {amountTransactionCompletedSuccessfully (old
    marketBalanceOf) marketBalanceOf (old importBalanceOf)
    importBalanceOf !market _to}
  =
  amount_transaction (marketBalanceOf) (importBalanceOf) (!market) ( _to) (_value);
  add_gas (transferFrom_gas_consumed)

end

module ETPMarketBis

use int.Int
use my_library.SmartMeterID
use my_library.Address
use my_library(UInt256
use my_library.Uint
use Gas
use ETPMarket
use ETPAccount
use ETPRegistry
use ETPRegistryBis
use ref.Ref
use Trading
val onlyAlgo : ref bool (*modifier*)
constant mcomplete_gas_consumed : gas

axiom mcomplete_gas: mcomplete_gas_consumed ≥ 0

(* private function *)
let eTPMarket_complete (sellId: Peano.t) (buyId : Peano.t) (_purchase : purchase) : unit
  requires {!onlymarket}
  requires { (_purchase.amount_p) > 0 ∧ (_purchase.price_p) > 0 }
  requires { (Bal.([]) marketBalanceOf !market) > 0 }
  requires { acceptableAmountTransaction marketBalanceOf importBalanceOf !market ((Ord.([]) buyOrd.ord buyId).orderAddress) _purchase.amount_p}
  requires { acceptableEtherTransaction balance (Ord.([]) buyOrd.ord buyId).orderAddress (Ord.([]) sellOrd.ord sellId).orderAddress ( _purchase.price_p)}
  requires {!onlyAlgo}
  requires { sellId ≥ 0 ∧ buyId ≥ 0 }
  requires { Ord.mem_ sellOrd.ord sellId}
  requires { Ord.mem_ buyOrd.ord buyId}
  requires { uniqueAddress (Ord.([]) sellOrd.ord sellId). orderAddress (Ord.([]) buyOrd.ord buyId).orderAddress}
  ensures { etherTransactionCompletedSuccessfully (old balance) balance (Ord.([]) buyOrd.ord buyId).orderAddress (Ord.([]) sellOrd. ord sellId).orderAddress}
  ensures { amountTransactionCompletedSuccessfully (old importBalanceOf) importBalanceOf (old marketBalanceOf) marketBalanceOf (Ord.([]) buyOrd.ord buyId).orderAddress !market} =
  let sellOrder = Ord.([]) sellOrd.ord sellId in
  let buyOrder = Ord.([]) buyOrd.ord buyId in
  eTPAccount_complete (sellOrder.orderAddress) (buyOrder. orderAddress) (_purchase.price_p);
  transferFromMarket (buyOrder.orderAddress) (_purchase.amount_p);
  add_gas (mcomplete_gas_consumed)
end

Appendix B : WCET of function with allocation

type list α = Nil | Cons α (list α)
function length (l: list α) : int =
match l with
| Nil → 0
| Cons _ r → 1 + length r
end

let rec length_ [evm:gas_checking] (l:list α) : int32
requires { (length l) ≤ max_int32 }
ensures { !gas - old !gas ≤ (length l) * 128 + 71 }
ensures { !alloc - old !alloc ≤ 0 }
ensures { result = length l }
variant { l } =
match l with
| Nil → add_gas 71 0; 0
| Cons _ l → add_gas 128 0; 1 + length_ l
end

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

let g_ [evm:gas_checking] (i:int32) : int32
requires { 0 ≤ i }
ensures { !gas - old !gas ≤ i * 313 + 242 }
ensures { !alloc - old !alloc ≤ i * 96 + 32 } =
add_gas 58 0;
let l = mk_list42 i in
length_ l