The Conceptual Schema of Ethereum and of the ERC–20 Token Standard

Antoni Olivé

Department of Service and Information System Engineering
Universitat Politècnica de Catalunya – Barcelona Tech, Barcelona, Catalonia
antoni.olive@upc.edu

Abstract. There is an abundant literature on Ethereum, but as far as we know what is missing is its explicit conceptual schema. We present here the conceptual schema of Ethereum in UML. The schema should be useful to those that want to understand Ethereum. We also show that the schema is necessary for developing the schema of Ethereum–based DApps. We present a few population constraints, and show that they suffice for the specification at the conceptual level of what is understood by immutability of a blockchain. We also show that the well–known reification construct and an initial constraint suffice to specify at the conceptual level that the Ethereum blockchain stores the full state history.

Keywords. Conceptual modeling, Conceptual schema, Blockchain, Ethereum, DApps, Immutability.

1. Introduction

This paper reports the main results of a project aiming at developing the conceptual schema of Ethereum, a popular open-source platform for blockchain-based decentralized applications [1]. The project had two main goals: (1) to know the conceptual schema of that system, and (2) to check the degree to which the constructs that have been developed in the conceptual modeling field allow the complete specification of a complex system like Ethereum. Of particular concern was how to specify immutability at the conceptual level.

The rationale of the first goal was that so far most of the Ethereum literature is written from either a technical or an economic perspective [2]. Application developers, researchers and students in general that need to learn the foundations of Ethereum have easily available a large number of books, papers and web documents (such as, for example, [3,4,5]), but they usually include (and, sometimes, focus on) many complex implementation details that make their understanding difficult [6].

From a conceptual modeling point of view, it is easy to see that what is missing in the above literature is the conceptual schema. Ethereum, like all blockchains, is basically a particular kind of distributed database [7,8] and, as such, it necessarily has a conceptual schema. The important role of the explicit definition of that schema not only in the development of database and of information systems, but also in their understanding, has been recognized since long ago [9,10,11].

The rationale of the second goal was that blockchains in general, and Ethereum in particular, have some features whose conceptualization is not obvious. We wanted to check whether the constructs provided by conceptual modeling languages are sufficient to deal with those features. One of them, which is present in all blockchains, is *immutability* [12]: what kind of integrity constraints are needed to specify immutability? The other feature, which is specific to Ethereum, is that, besides the transactions, it maintains the *full state history* of the state of the instances of *Account*, which is the main entity type represented in the blockchain. The question is then: do we need a temporal conceptual model [13,14,15] to specify the full state history?

We describe here the main result of our project: the conceptual schema of Ethereum in UML. We deal only with the main elements of the structural schema; the behavioral one, at the conceptual level, is simpler. We have found that standard UML, extended with a few known temporal constraints, suffices for defining that schema, including the blockchain immutability and its full state history. We hope that the schema will be useful to those interested in learning and using Ethereum. On the other hand, we show that the schema is necessary for the development of the conceptual schema of Ethereum–based applications.

The structure of the paper is as follows[1]. Next section introduces and formally defines the temporal constraints that will be needed. Section 3 presents the conceptual schema of Ethereum. Section 4 shows the role of that schema in the development of decentralized applications based on the ERC20 standard. Section 5 reviews related work. Section 6 summarizes the paper and suggests further work.

2. Population and initial constraints

In this section, we define the temporal constraints that will be used in this paper. These constraints have been previously presented in the literature using several terms and formalisms [16,17,18,19]. We use here the terminology of the temporal constraints defined in [20], give their formalization in first-order logic, and indicate how to use the constraints as stereotypes in UML. Figure 1 shows several examples of the constraints introduced here.

We adopt a logical and temporal view of the information base, and assume that entities and relationships are instances of their types at particular time points (or states). By *lifespan* we mean the set of times during which the system operates. We represent by *Time*(t) the fact that t is a time point of the lifespan. In the information base, if t is a time point of the lifespan, we represent by $E(e,t)$ the fact that e is an instance of entity type E at t. In this paper, we assume that entity types have a single existence interval [17], that is, if an entity e ceases to be instance of a type E it cannot be instance of it later on.

[1] This paper is an extended version of [24].

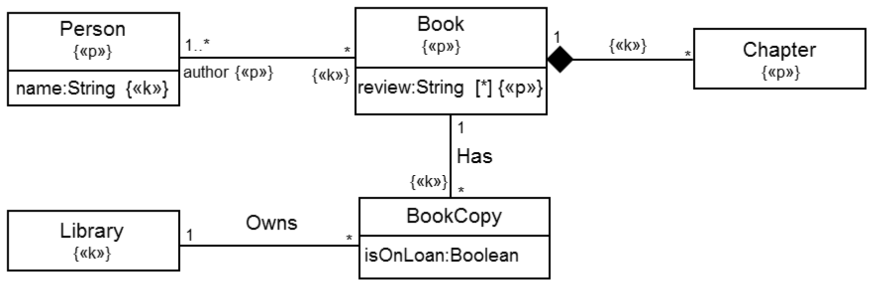

Fig. 1. Examples of population constraints

We denote by $R(p_1{:}E_1,...,p_n{:}E_n)$ the schema of a relationship type named R with entity type participants $E_1,...,E_n$, playing roles $p_1,...,p_n$, respectively. When the role name is omitted, it is assumed to be the same as the corresponding entity type. In the logical representation, attributes will be considered as ordinary binary relationship types. In the information base, if t is a time point of the lifespan, we represent by $R(e_1, ...,e_n,t)$ the fact that entities $e_1,...,e_n$ participate in a relationship instance of R at t.

2.1 Entity type population constraints

The *population* of an entity type E is the set of its instances at some time (or state). An entity type is *constant* when its population is always the same. Formally[2]:

$E(e,t) \rightarrow \forall t_1\, (Time(t_1) \rightarrow E(e,t_1))$

An entity type E is *permanent* when once an entity e becomes an instance of E, e continues to be an instance until the end of the lifespan. Formally:

$E(e,t) \rightarrow \forall t_1\, (Time(t_1) \wedge t_1 > t \rightarrow E(e,t_1))$

It can be seen that a constant entity type is also permanent. On the other hand, if E is a covering generalization of a set of permanent entity types, then E is also permanent.

In UML, the above constraints can be defined as stereotyped constraints to which we give the short names of k (for constant) and p (for permanent).

In the example of Fig.1, it is assumed that the set of libraries is always the same, and therefore *Library* is constant. *Book* and *Chapter* are permanent because once a book is published it exists forever. It is assumed that once an instance of *Person* is created, it is known to the system forever. The population of *BookCopy* is unconstrained because its instances may cease to exist (withdrawn, lost, etc.).

[2] For simplicity, we assume that all free variables are universally quantified in front of the formula.

2.2 Relationship type population constraints

The *population* of a relationship type R is the set of its instances (relationships) that exist at some time (or state). We say that a relationship type $R(p_1{:}E_1,\dots,p_n{:}E_n)$ is *constant* with respect to a participant p_i if the instances of R in which an instance e_i of E_i participates are the same during the temporal interval in which e_i exists. Formally:

$$R(e_1,\dots,e_i,\dots,e_n,t) \rightarrow \forall t_1\,(E_i(e_i,t_1) \rightarrow R(e_1,\dots,e_i,\dots,e_n,t_1))$$

Similarly, R is *permanent* with respect to participant p_i if the instances of R in which an instance e_i of E_i participates never cease to exist while e_i is an instance of E_i. Formally:

$$R(e_1,\dots,e_i,\dots,e_n,t) \rightarrow \forall t_1\,(E_i(e_i,t_1) \wedge t_1 > t \rightarrow R(e_1,\dots,e_i,\dots,e_n,t_1))$$

A relationship type R is *constant* if it is constant with respect to all its participants. Similarly, R is *permanent* if it is permanent with respect to all its participants.

It can be seen that a constant relationship type is also permanent.

In UML, the above constraints can be defined as stereotyped constraints to which we give the same short names as before: k (for constant) and p (for permanent).

In the example of Fig.1, the composition *book–chapter* is constant because (we assume that) the chapters of a book are known when the book is created and they are always the same. Similarly, the book of a chapter is known when the chapter is created and it is always the same. The role *author* in the association *author–book* is permanent because a person may be an author of more books during its existence.

In the association *Has*, *bookCopy* is constant because the book is known when the book copy is created, and it is always the same. In that association, the role *book* is unconstrained because existing book copies may cease to exist while others can be added. The association *Owns* is unconstrained if we assume that the library that owns a book copy may change during its existence.

Figure 1 shows also three examples of attributes. The *name* of *Person* is constant, the set of *review*s of a book is permanent (reviews can be added but not deleted) and attribute *isOnLoan* of a book copy is unconstrained.

2.3 Creation-time constraint

A creation-time constraint ϕ of an entity type E is a constraint that its instances must satisfy only at the time when they become an instance of E [21]. Formally:

$$(E(e,t) \wedge \neg\exists t'(t' < t \wedge E(e,t'))) \rightarrow \phi(e,t)$$

In the example of Fig.1 a creation time constraint is that when a book is created there is at least one book copy. That is, the creation of a book must be simultaneous to that of at least one of its book copies; later on, new copies can be added and existing copies may cease to exist. Formally:

$$\phi(b,t) \equiv \exists bc\,(BookCopy(bc,t) \wedge Has(b,bc,t))$$

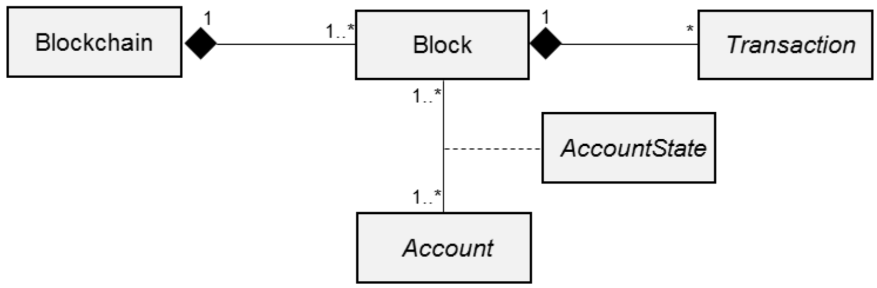

Fig. 2. Main concepts of the conceptual schema of Ethereum

3.Ethereum

Figure 2 is a broad view of the main concepts of the conceptual schema of Ethereum in UML[3]. The *Blockchain* consists of a set of *Block*s, which in turn consist of a set of *Transaction*s. The state of the system consists of a set of *Account*s and their properties. Transactions change that state. For each block, the system stores the state of the accounts (*AccountState*) at the moment when the transactions included in the block have been processed and the block has been added to the blockchain. In what follows we describe in detail those concepts.

3.1 Accounts

There are two kinds of accounts: Externally Owned Account (abbreviated as *EOAccount*) and *ContractAccount*, see Fig. 3. Both are permanent. Their generalization is the abstract entity type *Account*, which is also permanent.

Accounts are identified by means of their *address*, which is a constant attribute. An externally owned account is created and controlled by a user. Its address is determined from the user public key, which in turn is determined from the user private key. The sets of private/public keys of the users are stored in their wallets (not shown in the Figure).

A contract account is controlled by the code it contains, and its address is assigned by the system when the account is created. A contract account can be created by a user or by the code of another contract account.

[3] In the figures, greyed rectangles denote entity types whose complete definition is shown in other figures.

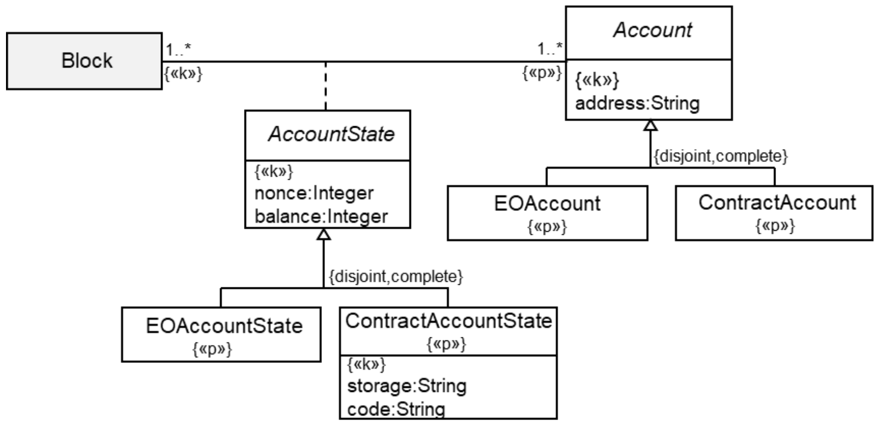

Fig. 3. Accounts and their states in Ethereum

Besides their address, both kinds of accounts have two attributes, called *nonce* and *balance*. For externally owned accounts, attribute *nonce* indicates the number of transactions sent from them, while for contract accounts it indicates the number of contract–creations made by them.

Attribute *balance* indicates the amount of *ether*, the cryptocurrency of Ethereum, owned by the account. The balance is represented in wei, the smallest subunit of ether.

Attributes *code* and *storage* apply only to contract accounts. Attribute *code* contains the code that is executed when called by a transaction or by another contract account. The code is written in the EVM code language, and it is executed by the Ethereum Virtual Machine. Normally, the code of an account cannot change, but there is the possibility of executing a *destruct* operation with the effect that the code and the storage are removed from the account. Note that if *code* could not be destructed, then we could define it as a constant attribute of *ContractAccount*.

Contract accounts have also an attribute called *storage*, which is used by the contract code. A contract can neither read nor write to any storage apart from its own.

In Ethereum, there are two kinds of states: world state and account state. An *account state* is the set of values of the attributes of an account at a given moment. A *world state* is the set of all accounts existing at a given moment and their account state at that moment.

For each block, Ethereum stores the world state at the moment when the transactions included in a block have been processed and the block has been added to the blockchain. Therefore, the world state of a given block includes all accounts and their state existing after all transactions included in the block have been processed.

The world and the account states have been modeled in Fig. 3 by the association between *Block* and *Account*, and its reification, the entity type *AccountState*. Both the association and *AccountState* are permanent. In the association, the role *block* is constant, meaning that the set of accounts to which a block is associated with is fully determined when the block is added to the system, and cannot be changed. The role *account* in that association is permanent because new instances can be added at any time.

For each block, there is an instance of the association *block–account* (and therefore of *AccountState*) for each account that exists at the time the block is created. This can be easily expressed by means of a creation-time constraint (see sect 2.3). In logic, if *R* is the association *block–account*, the constraint would be:

$\phi(b,t) \equiv \forall a\ (Account(a,t) \rightarrow R(b,a,t))$

Note that given that *Account* is permanent, once an account is created, it will be associated with the block within which it was created and with all future blocks.

An instance of *AccountState* is an account state of the corresponding account. There are two permanent subtypes of *AccountState*, *EOAccountState* and *ContractAccountState*, similarly to the two subtypes of *Account*. The account attributes have been defined in these entity types, and all of them are constant.

The current values of the account attributes could have been defined as derived attributes of *Account* and of *ContractAcccount*. These attributes would not be constant. However, for simplicity, this has not been done in Fig. 3. The derivation rules would indicate that their value is that of the *AccountState* or *ContractAccountState* instances corresponding to the same account in the last block.

The set of instances of *AccountState* of a block is the world state corresponding to that block. Given the population constraints of *AccountState*, *Account*, *Block* (we will see that is also permanent) and those of the association *block–account* it follows that Ethereum stores the full history of its states.

With respect to immutability, the schema fragment of Fig. 3 indicates that the instances of *Account* and *AccountState* cannot be deleted and their attributes cannot be modified. Moreover, the instances of the association *block–account* of a block cannot be changed. However, and this is a subtle and necessary point, it is possible to add instances of that association to accounts.

3.2 Transactions

There are two kinds of transactions: *MessageCall* and *ContractCreation*, see Fig. 4. Both are permanent. Their generalization, the abstract entity type *Transaction*, is also permanent. All attributes of transactions are constant.

Transactions can be identified in three ways. The first is by means of attribute *id*, which is automatically computed when the transaction is created. The second is the tuple (*sender*, *nonce*). Transactions are originated by externally owned accounts, which send them to the network for processing. The association *Sends* indicates the *sender* of a transaction. The association is permanent, with role *sender* permanent (an account can send several transactions) and role *transaction* constant (the sender is determined when the transaction is created and cannot be changed later). Transactions sent by an externally owned account are numbered consecutively (*nonce*) starting at zero. The third way of transaction identification involves attribute *index*, which will be explained in the next section.

A message call is a transaction sent to a recipient. The association *Receives* indicates the *recipient*. The association is permanent with role *messageCall* constant and role *recipient* permanent. If the recipient is an externally owned account, the ether

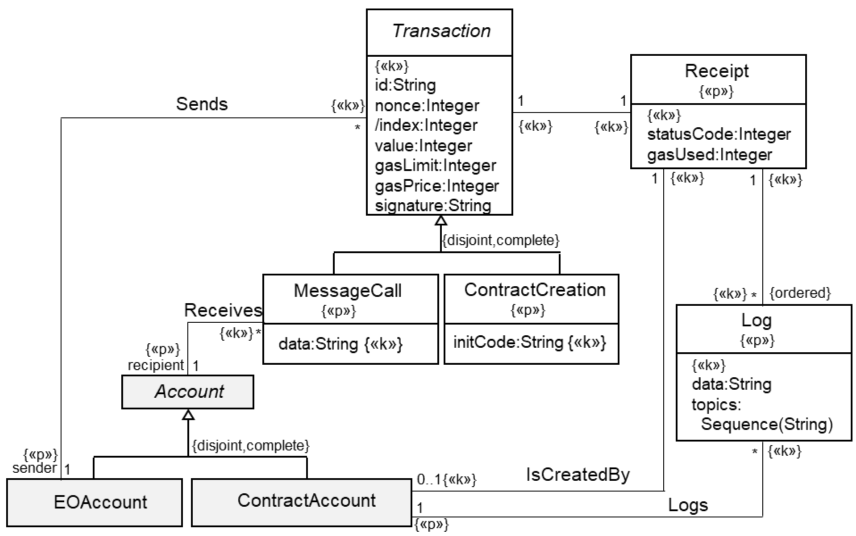

Fig. 4. Transactions in Ethereum

indicated in the attribute *value* is transferred from the sender to the recipient. If the recipient is a contract account, then its *code* is executed using the transaction attribute *data* as input. In this case, the transaction *value* may or may not be transferred to the recipient account.

A contract creation is a transaction that creates a new contract account. The *code* of the new account is obtained from the *initCode* attribute and it can be executed in future message calls. The transaction *value* is the starting balance of the new account (may be zero).

In order to ensure that a transaction has been originated by the sender, the transaction includes a *signature* attribute. The signature is obtained from the private key of the sender and the transaction attributes. Given a transaction, anyone can check that only the owner of the sender account could have sent it.

In Ethereum, processing a transaction has a fee, which is paid by the sender. The fee is expressed in units of gas. A unit of gas has a price in ether. When the recipient of a transaction is a contract account, it may be difficult to know in advance the amount of gas to be spent in a transaction. In order to control the maximum fee to be paid, transactions include attributes *gasLimit* and *gasPrice*. The first indicates the maximum number of units of gas to be spent, and the second the price of each unit of gas the sender is willing to pay.

Once a transaction has been executed, Ethereum generates a *Receipt* that encodes information from the transaction execution. In the association *transaction–receipt* both roles are constant. *Receipt* has two constant attributes: *statusCode,* which indicates whether the transaction has been successful or a failure, and *gasUsed*, which is the amount of gas used by the transaction. If the transaction was a

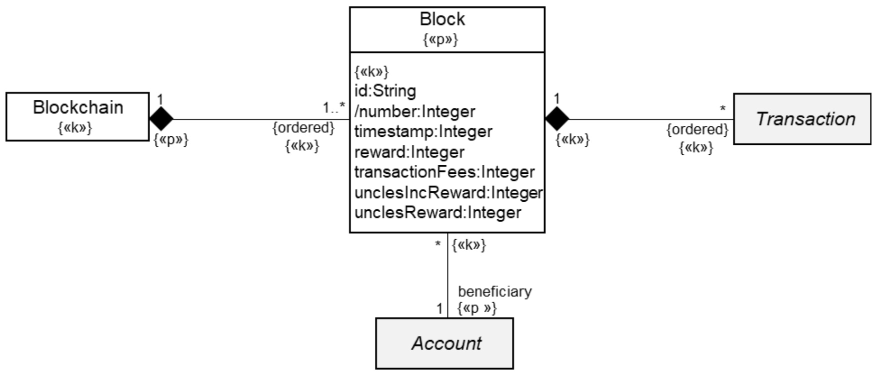

Fig. 5. Blocks in Ethereum

ContractCreation, then the association *IsCreatedBy* relates the contract account with the receipt of the transaction that created it. The two roles of the association are constant.

During the execution of a transaction, the code of the contract accounts involved in that transaction can add entries to the log of the transaction. *Log* is permanent and has two constant attributes: *data* and *topics*. The meaning of these attributes is application dependent. The two roles of the association *log–receipt* are constant. In the *Logs* association, *contractAccount* is permanent while *log* is constant.

With respect to immutability, the schema fragment of Fig. 4 states that the instances of *Transaction*, and its subtypes, *Receipt* and *Log* cannot be deleted and their attributes cannot be modified. Three associations (*transaction–receipt*, *log–receipt* and *IsCreatedBy*) are constant, meaning that their instances cannot be deleted and no new instances can be added to the existing participants in those associations. The other three associations (*Sends*, *Receives*, *Logs*) are permanent, which implies that their instances cannot be deleted, but it is possible to add new instances to entities with a permanent role.

3.3 Blocks

In Ethereum, the blockchain consists of an ordered sequence of blocks. Figure 5 shows the entity types *Blockchain* and *Block* and the composition association between them. *Blockchain* is constant, and its population consists of a single instance, while *Block* is permanent and its population consists of many instances. The role *blockchain* is permanent because new blocks are added to the composition, while the role *block* is constant because a block is associated to the blockchain when it is created and cannot be changed.

All attributes of *Block* are constant. The first two are identifiers of blocks. Attribute *id* is a hash computed by the system from the block's contents, which includes several attributes irrelevant to our conceptual modelling purposes. Attribute *number* is derived. The corresponding derivation rule defines its value as the index of the block in the composition. The first block has a number of zero.

As has been indicated, the role *block* in the blockchain composition is constant. However, in this case the role *block* is *ordered*, which means that the blocks of the blockchain are ordered (a sequence in this case). This raises a subtle point: what precludes the change of the order of the blocks in the sequence? We could define a new population constraint for this purpose but in this case it is not necessary. It suffices to define attribute *number* as constant, which implies that the position of a block in the sequence cannot change.

Attribute *timestamp* indicates the time when the block was added to the blockchain. An obvious constraint is that it must be greater than that of the previous block in the sequence.

Blocks are prepared and added to the blockchain by *miners*, which are specialized network nodes. The miner of a block is compensated (in ether) for the work done. The compensation is sent to an account designated by the miner, given by the association *block–beneficiary* in Fig. 5. The compensation includes a reward (attribute *reward*) and the fees of all transactions included in the block (attribute *transactionFees*). In some cases, a block may include up to two special stale blocks, called uncle blocks, which do not include transactions. If it is so, then the *beneficiary* receives an additional reward (attribute *unclesIncReward*), and the uncles receive the reward given by attribute *unclesReward*. For simplicity, Figure 5 shows neither the uncle blocks nor their beneficiaries.

A block, in turn, consists of an ordered sequence of transactions. Figure 5 shows the composition association between *Block* and *Transaction*. Note that both roles in that association are constant: the instances of the composition are determined when a block and a transaction are created and cannot be changed. A block and its transactions are recorded in the blockchain at the same time.

In addition to the two ways indicated in the previous section, an instance of *Transaction* can be identified by the block of which it is a part and the *index* attribute (Fig. 4). This is a derived attribute whose value is the position of the transaction in the block. The attribute is constant, which –among other things- means that the position on a transaction in the block cannot be changed.

With respect to immutability, the schema fragment of Fig. 5 indicates that the single instance of *Blockchain* exists since the beginning of the system's lifespan and, as well as the instances of *Block*, it cannot be deleted. The attributes of both types cannot be changed. One association (*block–transaction*) is constant. The other two are permanent. It is possible to add blocks to the blockchain (association *blockchain–block*) and to add blocks to an account (association *block-beneficiary*). Changing the position of a block in the blockchain or the position of a transaction in a block is not allowed.

It is interesting to see that the population constraints allow us to easily define that it is possible to add blocks to the blockchain, but it is not possible to add transactions to a block.

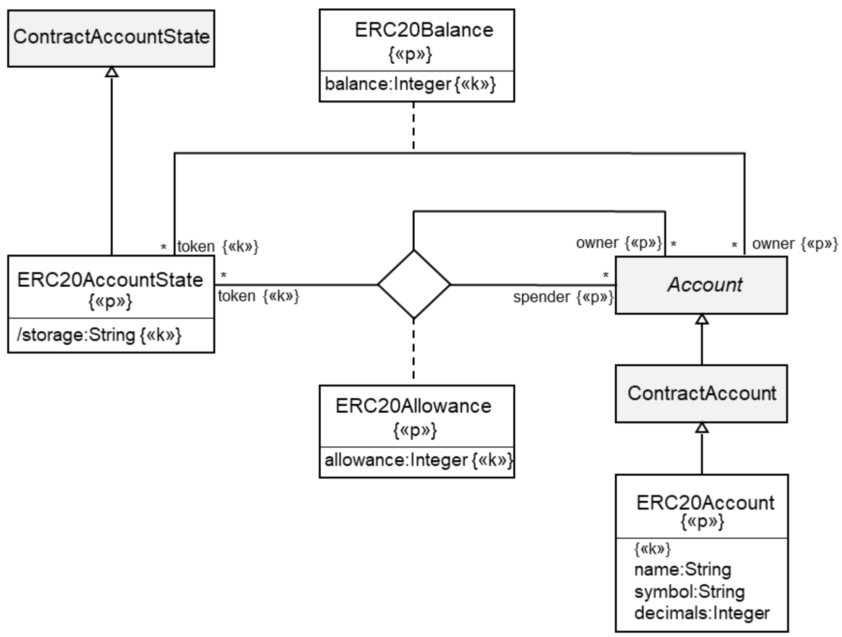

Fig. 6. ERC20 accounts

4. The ERC Token Standard

Ethereum is a platform for the development of decentralized applications, or DApps. Among other things, a DApp accesses and stores information in the blockchain, and invokes the execution of contracts stored in contract accounts. Therefore, the schema of a DApp must *necessarily* include (in UML, import) elements defined in the Ethereum schema.

As an illustration of the relationship between both schemas, in this section we deal with a popular family of DApps: those implementing the ERC20 token standard [23]. In the standard, a token is something that is owned by accounts and can be transferred between accounts. ERC20 is a standard for fungible tokens, which are those that are interchangeable as, for example, private currency, shareholder equity, voting rights or hours in a time bank.

Figure 6 shows the UML schema of a DApp that strictly implements the standard. We show in grey the entity types from the Ethereum schema that would be imported.

Each token is managed by an instance of *ERC20Account*, a permanent subtype of *ContractAccount*. The instances have three constant attributes: *name* of the token (e.g. Euro), *symbol* (e.g. EUR) and *decimals*, which is the number of decimals used by the token (e.g. 2).

The state of an instance of *ERC20Account* is given by the above attributes and those common to all contract accounts, defined in *ERC20AccountState*, a permanent subtype of *ContractAccountState*. Two of them deserve special attention: *code* and *storage*. Attribute *code* contains the code that manages the corresponding token, which in general may be different from one token to another.

Attribute *storage* is an encoding of the *ERC20Account* attributes, and of the instances of *ERC20Balance* and *ERC20Allowance* (explained below) to which it is associated. For this reason, we have redefined it as derived.

The units of a token are owned by instances of *Account*, and can be transferred between accounts. Any account can be owner of tokens. The amount of units of a given token owned by an account is its balance. In Fig. 6, *balance* is a constant attribute of the permanent entity type *ERC20Balance*, which is the reification of the permanent association *token–owner*. An instance (*t*,*o*) of this association indicates that in the state corresponding to *t*, the owner *o* has a number of units (balance) of the token *t*. Note that *token* is constant, which means that the owners of a token, and their balances, are defined when an instance of *ERC20AccountState* is created, and they cannot be changed afterwards.

An owner of a token can delegate authority to a spender, allowing it to spend a specific amount (allowance) from the owner's balance. Both owner and spender are accounts. When the spender spends an amount on behalf of the owner, that amount is subtracted from both the owner's balance and the allowance. In Fig. 6, the delegation is modeled as a permanent ternary association involving a *token*, an *owner* and a *spender*. The reification of the association is the permanent entity type *ERC20Allowance*, whose only attribute is the *allowance*. Note that *token* is again constant, with the same implication as in *ERC20Balance*.

5. Related work

In the literature, the two works that are more related to ours are the blockchain domain ontology [2] and EthOn [6,22]. The blockchain domain ontology is not blockchain–specific, but general. It distinguishes three ontological layers (datalogical, infological and essential) and it includes an ontology for each layer. Our conceptual schema would basically be placed in their infological layer. The ontology corresponding to this level, the infological ontology, consists of six entity types, five associations, and one attribute. The conceptual schema that we have presented here is much more detailed because it is blockchain–specific.

EthOn is an ontology in RDF Schema and OWL that formalizes most of the concepts used in the Ethereum platform as described in the "yellow paper" [1]. The scope of EthOn is different from that of our conceptual schema. EthOn includes in an integrated ontology both the concepts related to the data stored in the platform and the concepts related to the implementation. In the classical terminology used in conceptual modelling [9], it can be said that EthOn describes in an integrated view both the conceptual and the internal schema of Ethereum. On the other hand, EthOn does not specify the population constraints of its concepts needed to specify their immutability, and it does not formalize the full state history.

6. Conclusions

We have presented the conceptual schema of Ethereum in UML. As far as we know, this is the first time that the schema is presented in the literature. We hope the schema will be useful to those that want to understand Ethereum. Moreover, we have argued that the Ethereum schema is necessary for developing the schema of Ethereum–based DApps. We have illustrated this in detail with the schema of a DApp implementing the ERC20 standard.

We have presented and formalized a few population constraints, and we have shown that they suffice for the specification at the conceptual level of what is understood by immutability of a blockchain. Finally, we have shown that the well–known reification construct and an initial constraint suffice to specify at the conceptual level that the Ethereum blockchain stores the full state history.

This work can be extended in several directions. We point out two of them here. First, it would be useful to complete the structural schema that we have presented with a few remaining minor details, and to develop the behavioral one. Second, a work similar to the one presented here could be done with other blockchain platforms like, for example, Bitcoin or Hyperledger.

Acknowledgments. The author is greatly indebted to Joan Antoni Pastor and Jordi Estapé for their comments to earlier drafts of this paper.

References

1. Wood, G. Ethereum: A secure decentralised generalised transaction ledger, https://ethereum.github.io/yellowpaper/paper.pdf (2020).
2. de Kruijff J., Weigand H.: Understanding the Blockchain Using Enterprise Ontology. In: Dubois E., Pohl K. (eds.) Advanced Information Systems Engineering. CAiSE 2017. Lecture Notes in Computer Science, vol. 10253. Springer (2017)
3. Antonopoulos, A.M., Wood, G.: Mastering Ethereum: Building Smart Contracts and DApps. O'Reilly Media (2018)
4. Dameron, M.: Beigepaper: An Ethereum technical specification. https://github.com/chronaeon/beigepaper/blob/master/beigepaper.pdf (2019)
5. Kasireddy, P.: How does Ethereum work, anyway? https://medium.com/@preethikasireddy/how-does-ethereum-work-anyway-22d1df506369 (2017)
6. Pfeffer, J.: EthOn — introducing semantic Ethereum. Organized Ethereum knowledge, https://media.consensys.net/ethon-introducing-semantic-ethereum-15f1f0696986 (2017)
7. Dinh, T.T.A., Liu, R., Zhang, M., Chen, G., Ooi, B. C., Wang, J.: Untangling Blockchain: A Data Processing View of Blockchain Systems, in IEEE Transactions on Knowledge and Data Engineering, vol. 30, no. 7, pp. 1366–1385 (2018)

8. Kim, H.M., Laskowski, M.: Toward an ontology-driven blockchain design for supply-chain provenance. Int. Syst. in Accounting, Finance and Management 25(1): 18–27 (2018)
9. ANSI: ANSI/X3/SPARC study group on data base management systems. Interim report. FDT, Bulletin of ACM SIGMOD 7(2) (1975)
10. Mylopoulos, J.: Conceptual Modelling and Telos. In: Loucopoulos, P, Zicari, R. (eds.) Conceptual Modelling, Databases and CASE, New York, John Wiley and Sons, pp. 49–68, (1992)
11. Delcambre L.M.L., Liddle S.W., Pastor O., Storey V.C.: A Reference Framework for Conceptual Modeling. In: Trujillo J. et al. (eds.) Conceptual Modeling. ER 2018. Lecture Notes in Computer Science, vol. 11157. Springer (2018)
12. Hofmann, F., Wurster, S., Ron, E., Böhmecke-Schwafert, M.: The immutability concept of blockchains and benefits of early standardization, 2017 ITU Kaleidoscope: Challenges for a Data-Driven Society (ITU K), Nanjing, pp. 1–8. (2017)
13. Gregersen, H., Jensen. C.S.: Temporal entity-relationship models-a survey, IEEE Transactions on Knowledge and Data Engineering, vol. 11, no. 3, pp. 464–497 (1999)
14. Combi C., Degani S., Jensen C.S.: Capturing Temporal Constraints in Temporal ER Models. In: Li Q., Spaccapietra S., Yu E., Olivé A. (eds.) Conceptual Modeling - ER 2008. ER 2008. Lecture Notes in Computer Science, vol. 5231. Springer (2008)
15. Artale A., Franconi E.: Foundations of Temporal Conceptual Data Models. In: Borgida A.T., Chaudhri V.K., Giorgini P., Yu E.S. (eds.) Conceptual Modeling: Foundations and Applications. Lecture Notes in Computer Science, vol. 5600. Springer, (2009)
16. Costal, D., Olive, A., Sancho, M.R.: Temporal features of class populations and attributes in conceptual models. In: Embley, D.W. (ed.) ER 1997. LNCS, vol. 1331, pp. 57–70. Springer (1997)
17. Cabot J., Olivé A., Teniente E.: Representing Temporal Information in UML. In: Stevens P., Whittle J., Booch G. (eds.) «UML» 2003 - The Unified Modeling Language. Modeling Languages and Applications. UML 2003. Lecture Notes in Computer Science, vol. 2863. Springer (2003)
18. Artale, A., Parent, C., Spaccapietra, S.: Evolving objects in temporal information systems. Ann. Math. Artif. Intell. 50(1–2): 5–38 (2007)
19. McBrien P.: Temporal Constraints in Non-temporal Data Modelling Languages. In: Li Q., Spaccapietra S., Yu E., Olivé A. (eds.) Conceptual Modeling–ER 2008. ER 2008. Lecture Notes in Computer Science, vol. 5231. Springer (2008)
20. Olivé, A.: Conceptual Modeling of Information Systems. Springer, Berlin (2007)
21. Olivé, A.: A method for the definition of integrity constraints in object-oriented conceptual modeling languages. Data Knowl. Eng. 59(3): 559–575 (2006)
22. Pfeffer, J.: EthOn: An Ethereum Ontology. https://consensys.github.io/EthOn/EthOn_spec.html. Accessed (March 2020)
23. Vogelsteller, F., Buterin, V. EIP 20: ERC–20 Token Standard. https:// eips.ethereum.org/ EIPS/eip-20 (2015)
24. Olivé, A.: The conceptual schema of Ethereum. In: Dobbie, G., Frank, U., Kappel, G., Liddle, S., Mayr, H. (eds.) ER2020 Conference proceedings (2020)