A Lightweight Approach to Smart Contracts Supporting Safety, Security, and Privacy

Olaf Owe and Elahe Fazeldehkordi

Department of Informatics, University of Oslo, Norway
{elahefa,olaf}@ifi.uio.no
March 24, 2022

Abstract

The concept of smart contract represents one of the most attractive uses of blockchain technology and has the advantage of being transparent, immutable, and corruption-free. However, blockchain is a highly resource demanding technology. The ambition of this paper is to propose a new approach for defining lightweight smart contracts, offering a high level of trust even without blockchain, when the underlying operating system can be trusted. Blockchain can be used for a higher degree of trust, for instance when the runtime system cannot be trusted. The approach gives transparency and immutability, and gives protection against corrupted or incorrect smart contract implementations. This is achieved by letting smart contract requirement specifications be separated from the smart contract implementations, provided by special objects, so-called history objects, recording all transactions of the associated contract. The history objects are generated by the runtime system as specially protected objects. Contract partners may interact with the history objects through predefined interfaces.

We present a framework which includes an executable, imperative language for writing smart contracts, a functional language for contract specifications by means of invariants over the transaction history of a contract, as well as a verification system. The framework allows compositional and class-wise verification. A history object can provide runtime checking of specified behavioral properties of the contract, and can provide safety, security, and privacy control, as well as trusted transfer of assets. We demonstrate the approach on an auction system.

Keywords: Transactions; Asynchronous Communication; Smart Contracts; Security; Privacy; Safety; Specification; Verification; Runtime Checking.

Contents

1. Introduction

Blockchain technology started initially with a new currency called Bitcoin that was based on automated consensus between networked users, not required to trust each other. Financial industries related to cryptocurrency have been seen as primary users of this technology and have resulted in the most widespread applications of blockchain, but its applications go far beyond financial ones. The concept of smart contracts represents one of the most attractive uses of blockchain technology that has appeared recently.

A smart contract is a program that executes the terms and conditions of an agreement that are predefined by mutually distrusting participants of the agreement. These programs are stored on blockchain, and their correct execution is enforced by the consensus mechanism of the blockchain without depending on a trusted authority. Compared to traditional contracts, smart contracts provide trust with low legal and traditional costs, no risk of tampering and fraud, no interference or trust issues of a third party. Apart from these advantages, smart contracts are automatic, fast, and transparent. The number of applications of smart contracts in industry and everyday life is countless. Applications include digital identity, banking, tax records, insurance, real estate, supply chains, IoT, gaming and gambling, auctions, authorship and intellectual property rights, life science, and health care.

Despite many advantages, there are also some drawbacks in smart contract technology. The concept of smart contracts is built on blockchain; therefore, it is expensive with respect to time, resources, and power consumption. Besides that, since it has the consensus mechanism of the blockchain at the bottom, privacy is not inherently supported. Our ambition is to suggest an approach that avoids these disadvantages.

In this work, we propose a new construct at the programming language level that supports some of the main advantages of smart contracts based on blockchain, including trust, immutability, and transparency, but is less expensive to implement since it is less dependent on the use of blockchain. It gives a level of trust at the application layer without use of blockchain. However, it can also be combined with the blockchain technology in order to improve the overall trust level on insecure platforms. Our approach offers predefined forms of security and privacy control and comes with a theory for formal specification and verification. In contrast to traditional smart contracts, our approach detects and protects against incorrect or tampered contract implementations.

More specifically, we propose a "container box" for recording all calls and futures (see Section 3.1), related to the interactions involving a given contract service provider. This "container box" will then record all transactions such as calls to/from the contract, as well as future values generated by the contract. For this reason, we will call it a history object. It can also be seen as a "ledger" since it records all the transactions reflecting communication with the service provider. The transaction history is generated by the underlying system, and programmers have only read access to history objects, controlled through interfaces. We associate one history object to each contract. We predefine a set of classes and interfaces for these history objects, restricting write access. These interfaces and classes cover safety, security, privacy aspects, as well as transfer of assets.

The need for formal verification of smart contracts is pointed out in several papers [3, 22]. Solidity is the most dominant language used for writing smart contracts; however, a main drawback with Solidity is that it is not well suited for specification and reasoning since it did not come with a formal semantics and is not oriented towards program reasoning [49]. For instance, class-wise verification is not supported, and even soundness can be a problem. In case of errors, Solidity uses roll-backs to return to a previous state, reverting all the modifications made until the last safe state. Reasoning about roll-backs is challenging since the cause of a roll-back can be implicit.

We therefore consider a high-level language that allows better reasoning support than Solidity. Our approach avoids the mentioned problems with Solidity. Furthermore, our language gives fewer runtime errors and less need for roll-backs, which simplifies reasoning. In particular, our approach supports compositional and class-wise verification, i.e., we can verify a class invariant by looking at the class itself and inherited code from superclasses, without looking at other classes. This is essential for scalability and open-ended program development, which are highly relevant factors for contracts.

In order to define history objects and contracts as autonomous distributed objects, our language is based on the active object paradigm [48]. This paradigm offers a natural and high-level understanding of service-oriented systems, and with a modular semantics, which is essential when we turn to specification and verification issues. Clients, contract, and history objects are then described by concurrent and distributed objects, with asynchronous and non-deterministic message passing. The language builds on the principle of interface abstraction, i.e., remote field access is illegal, and an object can only be accessed through an interface. Each object has one or more inter-

faces, and the only possible way of object interaction is through the methods defined in the corresponding interfaces. Our language combines first-class futures [7], which are often used in active object languages, and a restricted version of cooperative scheduling [14]. This novel combination gives flexible method interaction, scheduling control, simplified reproducibility of executions, support of roll-backs, as well as simplified verification. In particular, the support of roll-backs is due to the language restrictions on the use of guards. Furthermore, we show that a history object supersedes the functionality of the future mechanism. By providing a useful encapsulation context, the information in the futures are useful in a long perspective, and thus avoiding the need for garbage collection of futures. In addition, our solution improves the privacy and security control of future values.

We demonstrate our approach on versions of a smart contract example, based on the auction example of Ahrendt, Pace, and Schneider in [3]. We exemplify an active object defining an auction with an associated history object, and demonstrate how our approach can be used to provide aspects of security, privacy, safety, including high-level functional safety specifications that can be checked at runtime. We show how verification of the auction contract specification can be done in a simple manner by class-wise reasoning.

The main contribution of this paper is a framework for lightweight smart contracts based on the notion of history objects, consisting of an imperative language for contracts supporting roll-backs, an executable functional language for writing smart contract specifications over the transaction history, and a theory for class-wise verification, with language-based support of privacy, security, transfer of assets, and runtime checking of contract specifications. The history objects are part of the runtime system and provide trust even when the implementation of smart contracts and interacting parties are intentionally or unintentionally incorrect, or tampered with.

Outline. Section 2 describes relevant background on smart contracts and blockchain. Section 3 presents the main idea of history objects. Section 4 introduces an underlying programming and specification language, and Section 5 shows how history objects are implemented in this language. Section 6 focuses on contract specifications and safety, and section 7 discusses how to verify contracts, including verification of the example (Section 7.1). Sections 8, 9, and 10 deal with privacy, security, and trusted transfer of assets, respectively. Section 11 gives a comparison with Solidity and blockchain. The last two sections (12 and 13) present related work and a conclusion, respectively.

2. Smart Contracts and Blockchain

In systems with centralized control, parties who wish to trade with each other have to do this via the central system, which the parties should trust. Therefore, all the trust should be placed on the central system, and all the business transactions depend on this third party. This dependency could be costly for both parties of a transaction. Smart contracts came to solve these problems. The term smart contract was introduced by Szabo in 1997 [45]. It refers to simple programs that store rules for negotiation and terms of a contract. These terms will then be checked by a smart contract. Using smart contracts, untrusted parties can trade directly with each other. Smart contracts are stored on the blockchain, and each party has a copy of it.

A blockchain is a distributed ledger that is open to anyone; it stores the information across a network of computers. In general, a ledger is a list of records that can be in any form, like a notebook, or an excel file. In blockchain, a ledger is given by the complete information about all transactions of some kind, typically transactions with financial aspects. A distributed ledger is distributed across many locations instead of placing it in a fixed location. A blockchain is a chain of blocks. Each block holds some data together with the hash of the block, which is unique just like a fingerprint, and also the previous block's hash. The data stored inside the block depends on the type of blockchain; for instance, the Bitcoin blockchain stores details about the transactions, like the sender, the receiver, and the number of coins. Immediately upon creating a block, its hash is calculated. Any tampering inside a block will cause its hash to change, and therefore it makes the next block and all the following blocks invalid since they no longer store a valid hash of the previous block.

The use of hashes is not enough to prevent tampering, since computers nowadays can calculate hundreds of thousands of hashes per second, which means that someone can tamper with a block and calculate all the hashes of the other blocks again to make the blockchain valid. In order to mitigate the risk of tampering, a blockchain also uses a mechanism that is called proof-of-work, a mathematical computation that slows down the creation of the new blocks. This mechanism makes it harder to tamper with the blocks because if someone tampers with one block, he/she must calculate the proof-of-work for all the following blocks again. Therefore hashing and the proof-of-work mechanism provide trust in blockchain. Nevertheless, there is one more way that blockchains secure themselves, namely by being distributed.

Blockchain uses a peer-to-peer network, and everyone can join. When someone joins the network, he/she receives a full copy of the blockchain. The node uses this to verify that everything is in order. When someone creates a new block, that block will be sent to everyone on the network, each node can verify the block to make sure that it has not been tampered with, and if validity is accepted by the majority of the nodes, each node adds this block to their blockchain. All the nodes in the network create consensus, agreeing about which blocks are valid and which are not. This consensus promotes transparency and makes blockchains corruption-proof. Other nodes in the network reject those blocks that are tampered with. Therefore, without the consent of the majority of the nodes, no one is allowed to add a transaction block to the ledger. Besides, once a transaction block is added to the ledger, nobody can change it. So, no single user in the network can modify, delete, or update the blocks. This characteristic promotes immutability and makes sure that the blocks remain unchanged. For better space efficiency, one may store the hash value of a transaction rather than the transaction itself. This suffices for transaction validation, but not for extracting transaction details.

All the fundamental characteristics of the blockchain technology are also shared with smart contracts since smart contracts are based on the blockchain technology. Smart contracts can be applied to many different areas, not only exchange of money, but also property, stock, or anything else without having to go through a lawyer, a notary, or other centralized service provider. They entirely cut out the need for a middle man. Banks, for instance, can use smart contract to issue their loans or to offer automatic payments, insurance companies can use it to process specific claims, or postal companies can use it for payment on delivery. Other examples of smart contracts deal with escrow agreements, employment agreements, auctions, and voting systems.

Ethereum, the most prominent smart contract platform today, was first proposed in late 2013 by Vitalik Buterin [13]. Ethereum is an open software platform and is based on blockchain technology that enables developers to build and deploy decentralized applications. It focuses on running code for decentralized applications that deploy on its network, written in the form of smart contracts. Ethereum's blockchain not only allows currencies to reside on it but also software code. Parties or smart contracts in Ethereum can communicate with each other via transactions, in order to distribute assets between each other. In Ethereum, a smart contract is like an object in object-oriented languages such as C++ or Java. Objects in Ethereum are parties. Each party can have its own state and logic, similar to objects having

variables and methods in object-oriented languages, especially in the active object paradigm where the objects are distributed and autonomous and can have active behavior.

Several programming languages have been used for writing smart contracts on Ethereum like Solidity, Serpent, Go, and Lisp Like Language (LLL). LLL is similar to the Lisp language and was used mostly in the very early history of Ethereum and is probably the hardest to write in. Serpent is similar to the Python language and was popular in the early days of Ethereum, but the most popular and functional one currently is Solidity, which is very similar to JavaScript.

Solidity is a contract-oriented, high-level programming language. It is statically typed; it supports inheritance, libraries, and complex user-defined types, among other features. Solidity builds on Ethereum Virtual Machine (EVM), which executes an associated low-level bytecode language. This is similar to bytecode as used in the Java JVM or C# CLR. There are several compilers (e.g., SolC, a browser-based compiler) that compile smart contracts written in Solidity into EVM bytecode, which can then be deployed into the Ethereum blockchain and will be ready to receive transactions. So the entire life cycle of a smart contract in Ethereum is as follows:

$$Solidity \longrightarrow EVM\ bytecode \longrightarrow Deployment$$

In the EVM machine code, there are several operations. In Ethereum, each operation has a cost, in order to execute the smart contract, all the operations need to be paid. Ethereum has its own currency, which is named ether. Every transaction and execution of bytecode costs ether.

Gas, on the other hand, is a unit that translates into ether; for instance, if there are several instructions, the first instruction might cost two gas, and two gas get translated into some amount of ethers. The reason for separating gas and ether is to decouple the price of an operation with the market price of an ether. The gas price for an operation is constant and cannot easily be changed; however, the value of gas in terms of ether can be changed. A list of operation codes and how much each operation costs in terms of gas can be found in the Ethereum yellow paper (with the formal definition of the Ethereum protocol) [47]. In order to distribute assets between different parties in Ethereum smart contracts, each party sends ether via the transactions.

3. History Objects

We consider the setting of distributed concurrent objects communicating by asynchronous methods. A smart contract in this setting is reflected by an object providing a certain service to the environment. Such an object is called a contract object. We assume a smart contract supports a predefined interface Contract. An object is said to support an interface I if the class of the object implements I or a subinterface of I. Thus the interface *Contract* is the superinterface of all contracts. Consider a contract to be used by a number of clients. We propose to add an additional object, called the history object, associated with the contract, storing the history of transactions related to the contract, i.e., the messages corresponding to method calls and method returns. This object is provided automatically for each contract object, and records the transaction history. The history objects have a write-once read-many data structure, and are defined by a set of predefined generalized classes (say provided by a library). These classes are final in the sense that programmers may neither add fields, methods, nor method redeclarations. There is only one method, put, that changes the data structure, by adding a transaction to the transaction history. This method is called by the runtime system (whenever there is a new transaction) and is not visible to programmers. Therefore no object may manipulate the state of the history objects from source code.

For each contract object o we associate a history object:

$$o.history$$

The history object will keep track of all transactions involving the contract object. A contract participant may use a history object to check and verify that the interaction with the corresponding contract object is appropriate. From the outside, a history object is treated like a normal object, which means that any external object may communicate with a history object when desired, through method calls. This is following the spirit of [43].

Moreover, the behavior of the history objects is given by a set of predefined classes in combination with contract-dependent interfaces. The predefined classes allow a contract designer to select the trust level. In particular, there is a *get* method to retrieve information, something which enables a history object to take the role as a future (see Figure 1). The predefined classes redefine the *put* and *get* methods in order to provide protection of safety, security, privacy, and assets.

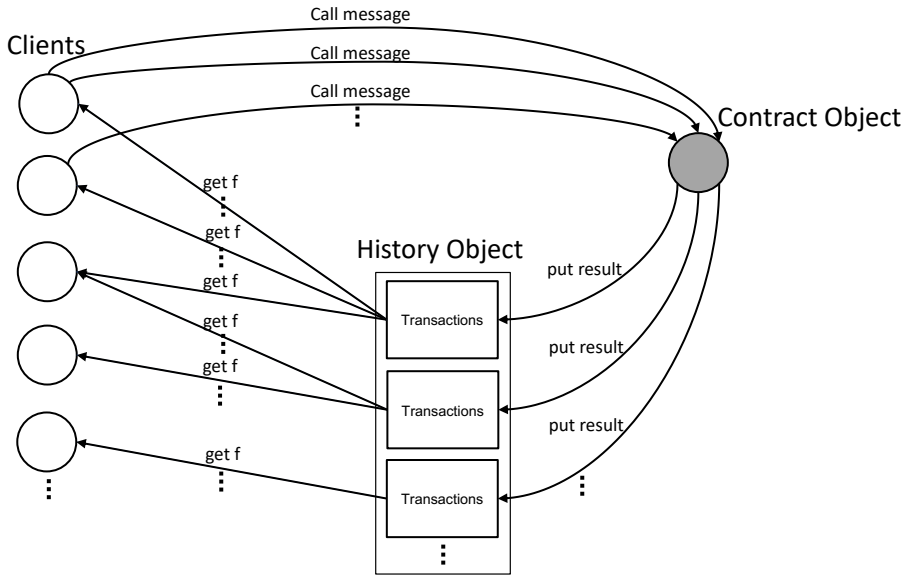

Figure 1: History objects and illustration of *call*, *put*, and *get* events

The history objects are provided by the underlying software/network platform. As long as this underlying software platform is protected from manipulation by attackers, the history objects can be trusted since they are not accessible through source code and therefore cannot be modified neither by attackers nor through intentional or unintentional programming "mistakes" in the contract object. Thus history objects are protected against application-level attacks. To improve trust, one can place the history object on a different physical location than the corresponding contract object, ideally with write-once read-many memory, or one may use a trusted platform if possible. In case the underlying platform cannot be trusted, one can use blockchain technology to ensure trust at the underlying platform level.

Our approach can be related to the future concept, which has become popular in the setting of active object languages and is supported by several languages [11]. Remote method calls are handled by message passing and the result of a method invocation is placed in a future object, at which time the future is said to be resolved. The caller generates a reference to the future object and this reference may be passed to other objects in the case of first-class futures. Any object with a reference to the future object may ask for the value, typically via a *get* statement, which will block when the future is not yet resolved. Some languages allow polling (i.e., testing repeatedly until

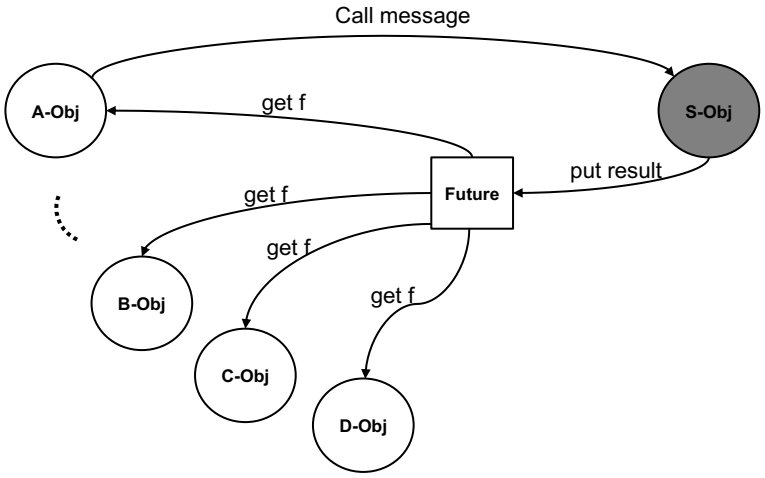

Figure 2: Illustration of the future mechanism

a condition becomes true) to check if a future is resolved, to avoid blocking. Figure 2 illustrates this use of futures.

A history object can be seen through the History interface, giving basic access to the transaction history, or through the more limited PreHistory interface, which provides the functionality of the future mechanism. These interfaces are given in Figure 3, using a type parameter I reflecting the interface of the associated contract. Furthermore, we provide subinterfaces of *History* adding safety, privacy, security, and transfer of assets, namely SafeHistory, PrivateHistory, SecureHistory, and AssetHistory, respectively. Note that the interfaces give read-only access. Implementation of these interfaces will be explained in Sections 6, 8, 9, and 10. The *PreHistory* interface has *put* and *get* methods to store and retrieve information, but such that the *put* method is not visible to programmers. Each call to, and return from, a method related to a contract generates a *put* call, which records this transaction event in the transaction history. When seeing a history object through the *PreHistory* interface one may use it as a future. By seeing it through the *History* interface one may access the whole or selected parts of the transaction history. The parameter *ctr* provides a reference to the contract object. The syntax for interfaces, the Transaction and Transfer types, and the other concepts used in the interfaces of Figure 3 are defined in Sections 4.1 and 4.5.

Behavioral specifications of interfaces will be given by means of invariants over the transaction history, denoted h, possibly involving user-defined functions over the history. The behavior of a contract is specified by invariants,

```
interface PreHistory[I] { // interface definition, generalized over interface I
  type Trans = Transaction[I]
  // Void put(Trans t) // used by the runtime system
  T get(Fut[T] f) // for each type T appearing as a method result in I
  // returns the future value (possibly error), when resolved.
}

interface History[I] (Contract ctr) extends PreHistory[I] {
  type Hist = List[Trans] // type Trans is inherited
  Trans lastTrans() // returns the last transaction
  Hist getTrans() // returns all transactions
  Hist transOf(Any o) // returns all transactions involving object o
  ... } // possibly other functions

interface SafeHistory[I] extends History[I] {
  func safe : Hist → Bool // defining legal histories
  invar safe(h) // an invariant over the transaction history h of the contract
  // built-in runtime check of safety as specified by "safe"
}

interface PrivateHistory[I] extends History[I] {
   func trusted: Any → Bool // defining which objects to trust
   // all transaction output to untrusted objects will be de-personalized
}

interface SecureHistory[I] extends History[I] {
  func blacklisted : Hist * Any → Bool
  // using blacklisting to control unserious parties
}

interface AssetHistory[I] extends History[I] {
  func cost : Trans → Transfer // specifying the cost of the transactions
  Int myBalance() // returns the accumulated balance of the caller
  Int contractBalance() // returns the accumulated balance of the contract
}
```

Figure 3: Generalized interfaces for history objects.

defining boolean conditions over the transaction history h. We therefore let the history h be available in specifications. In order to add runtime support of safety aspects, interface *SafeHistory* uses a boolean function safe defined over the history, such that $safe(h)$ is an invariant. The invariant $safe(h)$ will be checked at runtime. This is further explained in the class implementation of *SafeHistory* in Section 6.2, and an example is given in Figure 10. Note that *Any* is the superinterface of all objects.

Interface *PrivateHistory* will restrict the flow of private information to untrusted objects based on a function trusted, which is specified by the contract designer, reflecting a suitable policy for the contract. Interface *SecureHistory* uses a function blacklisted to detect harmful parties, denying access to blacklisted objects. Alternatively, one could use whitelisting, or define a function $secure : Hist * Trans \mapsto Bool$, expressing which transactions may be considered secure. This would allow more general control than blacklisting.

Finally, interface *AssetHistory* adds the aspect of transfer of assets, letting some or all of the transactions involve a cost. The *cost* function is specified by the contract designer for each transaction, depending on the particular contract. A history object supporting *AssetHistory* knows the balance of assets accumulated for each party including itself. A contract party may ask for his/her balance by method *myBalance*, and anyone may check the balance of the contract by method *contractBalance*.

As mentioned, we will give a high level implementation of these interfaces by predefined and protected classes that may not be redefined or extended by programmers or contract designers. However, contract designers may define suitable versions of the *safe*, *trusted*, *blacklisted*, and *cost* functions for their contracts, through subinterfaces of the given interfaces for history objects, and these subinterfaces can then be used in so-called "adaptations" of the predefined classes. The interface definitions are transparent to all users, but function definitions cannot be overwritten or redefined in any way in our language. Therefore, the combination of protected general classes for history objects and contract-specific functions gives trust at the software level.

Before showing how to implement the interfaces above, we will introduce a high-level programming and specification language (in Section 4).

3.1. Histories as a Generalization of Futures

With the standard future mechanism, it is not trivial to detect when a future can be discarded; and as many futures may be generated, garbage collection is in general needed. In the active object paradigm, this is a

clear disadvantage since the active objects themselves have a long life time. When local data inside objects is defined by data types, using a functional programming language to express and manipulate values of the data type (such as in the Creol and ABS languages), there is no need for general garbage collection of these values, assuming storage for values of the data types can be retrieved efficiently. In our proposed solution, the same history object will contain all future values generated by a given callee object, including those of the past as well as future ones. This history object is therefore long-lived and need not be garbage-collected. However, as the storage need is growing dynamically, the history objects could be placed on suitable storage media, possibly split in several parts, letting the latest part be most accessible.

Another disadvantage of the future mechanism is that the future value is unprotected, and an object getting the value may not know where the future came from and what it represents. In particular, privacy aspects are unknown and the information can easily be misused [32]. Our approach comes with the option of choosing safety and privacy restrictions (see Sections 6 and 8).

4. A High-Level Language for Active Object systems

Before defining history objects, we need an underlying language for interfaces, types, and classes – preferably one supporting distribution, autonomy, and compositional reasoning. We present a high-level language supporting active objects, based on the Creol/ABS concurrency model [27, 26]. In particular, we define a functional language for defining interfaces (Section 4.1), data types and functions (Section 4.2), and then an imperative language for defining classes supporting concurrent active objects (Section 4.3). Our language is strongly typed, and object variables are typed by an interface (not a class). In order to enable verification and specification of behavior, we build on language constructs for specification and reasoning supporting class-wise reasoning and interface abstraction [35, 36]. This means that the clients of a contract can rely on the abstract specification of the interface of the associated history object rather than program code. Furthermore, they may interact with the history objects if they do not trust the contract objects. The interfaces define visible methods and their specifications. A class may have several interfaces. Methods of a class that are not exported through an interface, are considered private.

4.1. Interface Definitions

An interface defines a view of a class object, in terms of methods, together with an invariant describing behavior and related data types and functions. We define the syntax of interfaces with an augmented Backus-Naur Form (BNF) regular expression:

```
interface I [[ I+ ]] [([T x]+)] [ extends I+ ] {
  [ [ with I ] [ T m([T x]*) ]+ ]*
  [ type and function definitions ]*
  [ invar R ]*
}
```

The superscripts * and + are used to denote repeated parts (* for zero or more and + for one or more repetitions), and the meta-symbols [ ] (without a superscript) are used to indicate optional parts. Optional class parameters (x of type T) are given by the syntax $([T\ x]^+)$. In order to allow generalized interface definitions, an interface may be parameterized by a number of types or interfaces using the syntax $[\,I^+]$ (where the brackets are part of the syntax). For example, **interface** $History[I]$ is generalized over I. We let I denote an interface name, T a type name, m a method name, x a formal parameter, and R an invariant assertion, which may refer to the contract transaction history h, and user-defined functions on the history. An invariant is an assertion that must hold initially and be maintained by every method.

The BNF interface definition shows how to define an interface I by extending a number of already defined interfaces (I^+), inheriting all definitions of these, and defining a number of methods, types, and functions. The functions may be partially or fully defined. Data types and function definitions are explained in the next subsection. A (directly or indirectly) extended interface is said to be a superinterface of I, and interface I is said to be a subinterface of its superinterfaces. The most general interface is called Any, and Contract is the superinterface of all contracts. Thus $Contract < Any$.

Note that a method may have a cointerface, given by a **with** clause, which defines the (minimal) interface of the caller object. The caller object is available inside a method body through the implicit parameter *caller*, typed by the cointerface. An object calling the method must be typed by a subinterface of the cointerface. A cointerface is needed when a method body is making calls back to the caller. In order to make these call-backs type-

```
interface Bidder {
  with Auction // only Auction objects may call the methods below
    Void newAuction() // inform a bidder that a new auction starts
    Void newBid(Nat x) // inform a bidder that x now is the highest bid
    Void youwon(Nat x) // inform the bidder that he/she won with bid x
    Void winner(Bidder o) // method to inform a bidder/owner that o won
    Void closed() // inform a bidder that the current auction is closed
}

interface Auction {
  Nat highest() // gives the highest bid in the current auction
  with Bidder // only Bidder objects may call the methods below
    Void open() // to open a new auction
    Bool close() // to close the current auction
    Bool makeBid(Nat x) // to place a bid in the current auction
}
```

Figure 4: Interfaces for the auction example

correct, we need to declare an interface for the caller (by a **with** clause). For instance, a method defined after the clause **with** *Bidder*, knows that the caller supports interface *Bidder* and is therefore allowed to make (type-correct) calls to methods of that interface.

Examples of interfaces for an auction system are given in Figure 4, and a cointerface is used. Interface Bidder defines the methods of bidder objects, and interface *Auction* defines the methods of the auction service provider. The cointerface used in interface *Auction* restricts clients calling open, close, and makeBid to Bidder objects (i.e., objects of classes implementing the Bidder interface).

Our language uses interfaces to describe distributed objects, which may have active behavior when desirable, and uses data types to define data structure local to an object. Since an interface declares no state variables, an interface can only talk about the transaction history h. Moreover, by means of functions defined over the history, one may express essential aspects of objects of the interface and define an abstract state. In a distributed, open-ended, and unpredictable environment, reasoning by means of history invariants is more suitable than reasoning based on pre- and post-conditions, since pre/post-conditions on the callee side will in general need to refer to

variables not found on the caller side (when these conditions need to talk about more than just the input/output). Our language includes executable history invariants, and allows runtime checking of invariants by means of history objects supporting *SafeHistory*.

We next define a functional language for defining data types and functions, and then use this to define transactions, histories, and specifications.

4.2. Data Type and Function Definitions

For the definition of recursive data types and partial functions, we use a syntax typical for strongly typed functional programming languages with pattern matching. We consider an executable functional language for data types and functions. As this language also is used for specifications, we obtain an executable specification language. A user-defined data type is defined by (named) disjoint unions $(... \mid ...)$ and products $(... * ...)$, allowing recursive definitions. A disjoint union may look like $c_1 : E_1 \mid c_2 : E_2 \mid ...$ where E_i is a type expression, often a product and c_i is implicitly defining a constructor function used to name an alternative in a disjoint union. Thus, a data type T is defined by a number of constructor functions for constructing T values. Each constructor function may have a number of input parameters. For instance, the list type could be defined by **type List**[T] = nil: | append: **List**[T] * T where *nil* and *append* are constructor functions. As *nil* has no input parameters, it is called a constant constructor. For convenience, we let the append constructor function be denoted by the infix symbol ";", and we omit writing an empty product. Generalized data type definitions are parameterized by one or more types or interfaces using the syntax [I+] as before. The list type can then be declared by the syntax:

```
type List[T] = nil: | _;_ : List[T]*T
```

where the underline indicates argument positions of functions with infix notation (and if needed, more generally for mixfix notations). The transaction history a, b, c can then be expressed as $nil; a; b; c$. The constructor functions define the set of possible values (as variable-free constructor terms) and are implicitly defined. Standard types such as Void, Bool, Nat, String are predefined with standard functions. The type Void is used to represent no information (i.e., a unit type), and *void* is the only value of this type. The language supports futures, and the predefined type **Fut**$[T]$ denotes the type of futures holding values of type T. The supertype of all types is called Data,

and the supertype of all interfaces is called Any. Thus *Any* is a subtype of *Data*. We let $\leq$ denote the subtype relation.

User-defined functions (other than constructors) are defined by a **func** declaration stating the input and output types of each function and by a number of equations where the left- and right-hand sides are expressions consisting of functions, constructor functions, and variables. The types of these variables are declared in a **var** clause. We restrict ourselves to equational function definitions (using multiset path ordering to avoid non-termination recursion [5]). We use the BNF syntax:

```
[ func g : T* → T ]* // function declaration
[ var [T x]*] // variable declaration
[ lhs = rhs [ if cond ] ]* // function definition
```

Here h is a function name, *lhs* (left-hand side) and *rhs* (right-hand side) are expressions, and *cond* is a boolean condition. A left-hand side defines a pattern and may contain the special symbols _ and **others** representing an arbitrary pattern and cases not covered, respectively. Any variable in a right-hand side must occur in the left-hand side.

Examples are given in Figure 5. The last *length* equation can also be written as $length(q; \_) = length(q) + 1$. We define the ends-with operator (**ew**), with infix notation using _ to indicate the argument positions as before, expressing that a list q ends with a given element x. A left-hand side may use others to match any other case not covered by the equations above. For instance, the last equation for **ew** could be replaced by the two equations $(q; x)$ **ew** $x = true$ and $(q;$ **others**$)$ **ew** $x = false$.

In order to deal with partial functions, we allow **error** in the right-hand side, for instance as in the definition of the above last function over lists. By static checking, one can ensure that a function definition properly covers all cases of possible inputs (without conflicting overlap). For instance, if the equation $last(nil) =$ **error** is omitted, the static checking would detect that $last(nil)$ is not defined.

4.3. Class Declarations for Active Objects

For the purpose of defining classes, we introduce a high-level language combining the active object paradigm, first-class futures, i.e., allowing futures to be passed like parameters to other objects, and guarded methods, i.e., identifying the conditions under which the methods are enabled [14]. We

```
func length : List[T] → Nat // list length
func _ew_ : List[T] * T → Bool // ends-with test
func last : List[T] → T // finds the last element, if any

var List[T] q, T x // variables used in the equations
length(nil) = 0
length(q;x) = length(q) + 1

nil ew x = false
(q;x) ew y = if x=y then true else false

last(nil) = error
last(q;x) = x
```

Figure 5: Examples of function definitions

define classes with the BNF syntax below:

```
class C [[ I+ ]] [([T x]+)] [implements I+] [extends C+] {
  [T w]* // fields
  [constructor body] // class constructor
  [ [ with I] [ T m([T x]*) {method body} ]* ]* // method definitions
  [ type and function definitions]* // as defined above
  [ invar R]* // invariant specification
}
```

In order to allow generalized class definitions, a class may by parameterized by a number of types, interfaces, and classes, using the syntax $[\,\mathsf{I}^+]$ as before. A class C may implement a number of interfaces and extend a number of superclasses. Thus we support multiple inheritance, but single inheritance suffices for our example. The body of the class definition (the part between "{" and "}") consists of field declarations, method definitions, possibly a class constructor method, as well as additional function, type, and invariant definitions. A class with an empty class body is called an empty class.

Any type, function, or parameter defined in an interface of a class is available in the class. Class parameters, fields, methods, as well as type and function definitions, are inherited in a leftmost, depth-first traversal of the

inheritance tree. A class must implement the methods of its interfaces and respect their invariant(s). For each method in an interface, the type of the parameters and cointerface (if any) must be a subtype of the types in the corresponding method definition in the class, and the result type in the class must be a subtype of that in the interface. A class invariant R may refer to class parameters and fields as well as the transaction history.

We allow an inherited name to be qualified by the superclass name in order to deal with inheritance of multiple definitions of the same name. For instance, if a class extends A, B, and both superclasses have a method m, then the subclass may refer to these as $m@A$ (or just m) and $m@B$. Superclass invariants and implements clauses are not inherited, which allows a subclass to freely redefine methods and invariants without the semantic constraints of the superclass. When desired, an invariant R of a superclass C can be inherited by stating **invar** $R@C$.

A class inherits the function and type definitions of its interfaces and superclasses. As we allow incomplete function definitions in interfaces, a class is allowed to complete the definitions of functions, but conflicting definitions are not allowed and checked statically. A class is executable if all functions are fully defined (i.e., the equations for function definitions are confluent and ground complete). Only executable classes may be used to create object instances. A contract designer can make executable adaptations of the predefined classes for history objects. Class adaptations have the syntax:

class C [**[** I^+ **]**] [([T x]$^+$)] [**implements** I^+] [**adapts** C^+] {}

An adaptation of a class is an empty subclass, instantiating the formal type parameters with actual ones. Thus a class adaptation may not add or redefine any methods or fields, but inherits definitions of functions from interfaces, in order to make the resulting class executable. In particular, we allow adaptation of final classes, and an adaptation of a final class is also final. By adaptation, a predefined non-executable class may result in an executable class with user-defined adaptations.

The predefined history object classes are final, which guarantees that the imperative code of these cannot be modified, while unspecified functions (such as *safe*, *trusted*, *cost*) can be defined by the contract designer, as desired for the particular contract. This means that a predefined history class may be specialized with different function definitions through different adaptations. In this way the methods of the predefined history classes are protected and cannot be changed by programmers, whether by purpose or by accident. This

is checked statically. Examples are shown in Figures 10, 12, 14, 15, and 17.

4.4. Method Definitions and Imperative Code

We consider a syntax inspired by the Creol/ABS language family. Let v denote a variable, o an object variable (an object reference), f a future variable (a reference to a future object), e an expression (assumed to be pure), and m a method. A variable is either a field w, a local variable y, or a formal parameter x (assumed to be read-only). We let this denote the current object, and a method has fid and caller as implicit parameters, identifying the future identity of the call and the caller object, respectively. We use capitalized words for types and interfaces, while variable and method names start with a lower-case character. Class names are written in upper-case characters.

A method body has the form:

when *guard*; $\overline{T\ y}$; *statements*

The body may have an (optional) initial guard, written **when** *guard*, in order to make sure the starting state is appropriate, otherwise the execution is delayed. The rest of the body consists of a number of typed local variables ($\overline{T\ y}$), followed by a statement list. Each branch of the body must have exactly one return statement, **return** e, where the value of the expression e is the resulting value, which must be of the method result type. The return statement is typically last, unless there is some local activity, like book-keeping, to take care of after the return. (A final return statement **return** *void* may be omitted).

The *guard* is either a boolean condition, which must be satisfied when the method starts, or the special construct $f?$, which checks that the future f is resolved. A **when** clause can be compared to the require statement in Solidity, but rather than resulting in a runtime error as in the case of require statement, a **when** clause will delay the method execution to a state where the guard is satisfied.

Methods with initial guards are expressive enough to avoid blocking calls and blocking *get* statements. The resulting value of a future generated by one method execution can be picked up in a guard by another method execution, either on the same object or on another object (which knows the future). Methods with an initial guard are semantically simpler than methods with internal guards, as suggested in [14]. The combination of first-class futures and initial guards has not been explored before.

We use the syntax **if** cond **then** s [**else** s] **fi** for if-statements (with optional then-part) and **while** cond **do** s **end** for while-statements. Assignments have the syntax $v := e$ where v is a variable and e a (pure) expression. We may abbreviate $T\ v; v := e$ to $T\ v := e$.

The statement syntax $v :{+}{+}\ e$ is permitted when v is a list, to express that an element e is appended at the tail of the list v (semantically, it is equivalent to the assignment $v := (v; e)$). This statement adheres to the write-once discipline since it cannot change previously written elements, and is used in the update transaction histories by the runtime system.

We consider various kinds of method calls. In order to avoid or control blocking calls, we combine one-way asynchronous calls, guards, and the future mechanism, which is popular in active object languages.

- An asynchronous call has the syntax $f := o!m(\overline{e})$ where o is the callee, m the called method, $\overline{e}$ the list of parameters, and where f is a future variable declared with type **`Fut`**$[T]$ where T is the return type of m. The future variable f is assigned a new call identity (a reference to the future object) uniquely identifying the call. This identity may be communicated to other objects. The caller is not blocked.

  In order to obtain the value returned from the call, the caller object (or another object that knows the future identity) performs $v :=$ **`get`** f where v is a program variable of type T. This statement will block if the future is not resolved, otherwise the result value is copied into v.[1]

- As a special case, an asynchronous blocking call has the syntax $v := o.m(\overline{e})$. Semantically, this is the same as $f := o!m(\overline{e}); v :=$ **`get`** f except that f is not visible in the program. The caller is blocked until the result is available, and then the result will be assigned to v. The caller and callee should be different objects, otherwise the statement may cause a deadlock.

- A local call has the syntax $v := .m(\overline{e})$, where m is an unguarded local method. The call is handled as an ordinary stack-based local call, and will not (in itself) cause a deadlock since there is no guard, but recursion may cause non-termination.

[1] When f is a future related to a contract c, the *get* statement $v :=$ **`get`** f is implemented as the asynchronous blocking call $v := c.history.get(f)$.

- A simple asynchronous call has the syntax $o!m(\overline{e})$. The caller is not blocked and cannot access the result value. It is used when the caller does not need the result value.
- A multicast has the syntax $list!m(\overline{e})$ where $list$ is a list of objects. The caller is not blocked, and the result values are not communicated to the caller. This is the same as making a simple asynchronous call $o!m(\overline{e})$ to each o in $list$.
- A broadcast has the syntax $I!m(\overline{e})$ where I is an interface. The caller is not blocked, and the result values are not communicated to the caller. A simple asynchronous call $o!m(\overline{e})$ is sent to all objects supporting I (possibly depending on network properties).
- An error handler can be appended to an assignment-like statement, including assignments, incremental assignments, synchronous calls, and *get* statements, using the syntax $< s >$ where s is the list of statements to be performed when the execution of the statement results in an error, If there is no handler, the current method returns an error. For instance $list : ++ last(q) < skip >$ will have no effect when the $last$ function returns an error (when q is empty). Without the handler, the method would result in an error. More advanced forms of exception handling would be beyond the scope of this paper.

The combination of futures and guards allows a programming style without blocking. For instance, the asynchronous blocking call to m in the code fragment $x := o.m(..); s$ where x is a field and s is the rest of the body, can be replaced by the non-blocking call to m in the fragment $f := o!m(..); this!n(f)$ with n defined as the guarded method:

$$Void\ n(\mathbf{Fut}[T]\ f)\{\mathbf{when}\ f?; T\ x := \mathbf{get}\ f; s\}$$

The guard ensures that the get command will succeed immediately. We here assume s is without local variables and return (local variables can be transmitted as parameters, and returns can be handled by delegation, using the mechanism of [35]).

The implementation of guarded methods involves a simple form of cooperative scheduling, letting each object have a method invocation queue. A method invocation is enabled when its guard is satisfied. When a method execution is completed, an enabled method invocation is selected from the queue. For instance, one may select the oldest enabled method invocation or

use some other queue ordering. Our language ensures that the history will determine the execution order since it determines all external inputs, and there is no other internal source of non-determinism, as captured by Theorem 1 below. An operational semantics for our language is provided in the appendix.

An example of a class defined in this language is given in Figure 6, showing a class implementing the *Auction* interface given in Figure 4. It defines class *AUCTION* with a number of fields and invariants. The invariants restrict the values of the fields of *AUCTION* objects between method executions. The example illustrates the use of the implicit parameter caller. The methods *open*, *close*, and *makeBid* use *Bidder* as a cointerface to restrict the callers to *Bidder* objects. This is needed for type correctness since the caller of *open* is assigned to *owner*, which is typed by *Bidder*, and method *close* sends a message to the owner object through the *Bidder* interface. A similar discussion applies to *makeBid* as well, whereas method *highest* need not have a cointerface since none is given in the interface. An implementation of a method must have the same (or wider) cointerface as in the interface.

Method *open* has a guard to ensure that there is no ongoing auction. This method completes when the auction object is ready for the new auction. Thus a blocking *open* call could cause a delay. An auction participant may instead do the non-blocking calls $ready := auction!open();\ this!myauction(ready); ...$, using a future *ready*. The *myauction* method can then take care of what to do when the auction starts and may be implemented with a guard as follows:

$$Void\ myauction(\mathbf{Fut}[Void]\ f)\ \{\mathbf{when}\ f?;\ \textit{actions related to the session}\}$$

As an improvement, one may redefine *makeBid* to ensure that the owner of an auction is not making bids on his own auction. Instead, we could give the owner a chance to set a minimal value as a parameter to *open*.

4.5. The Transaction Type Corresponding to an Interface

In order to define the transaction history, we consider the relevant kinds of events including invocations and completions, representing method call messages and return from a method, respectively, as well as get events reflecting the transfer of a future value in connection with a *get* statement or blocking call. These events are recorded in the history object. In addition, we may consider object generation events. In our framework, we do not need to consider reception of invocation messages (on the callee side) since these

```
class AUCTION implements Auction {
  Bool isopen:=false; // tells if the auction is open or closed
  Nat highBid:=0; // the current high bid
  Bidder highBidder; // the current high bidder
  Bidder owner; // the auction owner
  List[Bidder] bidders:=nil;

  invar isopen=(owner≠null)
  invar owner=null ⇒ highBidder=null
  invar highBidder=null ⇒ highBid=0

  Nat highest() {return highBid }

with Bidder
  Void open() {when not isopen; isopen:=true;
    owner:=caller; Bidder!newAuction()} // broadcast to all Bidder objects

  Bool close() { Bool ok:=(caller=owner);
    if ok then // only owner may close the auction
      if highBid > 0 then // isopen follows by the invariant
        owner!winner(highBidder); // see note on privacy
        highBidder!youwon(highBid); // to notify the winner
        bidders!closed(); // multicasting to bidders that the auction closes
        highBid:=0 fi;
      owner:=null; highBidder:= null; isopen:=false;
      bidders:= nil fi;
    return ok }

  Bool makeBid(Nat x){ bidders :++ caller;
    if open and x>highBid then
      highBid:=x;
      (bidders;owner)!newBid(x); // multicasting x to owner and all bidders
      highBidder:= caller; return true
    else return false fi }
}
```

Figure 6: The basic auction class

can be derived from the history. For our language, we therefore consider only invocation, and completion, and *get* events.

An invocation event corresponds to a call to a method m with actual parameter list $\overline{e}$ and is represented by a four-tuple of form:

$$call(u, caller, callee, m(\overline{e}))$$

where u is a unique identity generated for the call, a so-called future identity, *caller* is the caller object identity, and *callee* is the callee object. Note that the second and third arguments indicate the direction of the message (from-to). This event is generated when the corresponding invocation message is sent over the network. When the call has completed normally or abnormally (i.e., resulted in an error), the completion of the method by the callee generates the event:

$$comp(u, callee, caller, m(\overline{e}), result)$$

where *result* is the value resulting from the call, possibly **`error`**. As discussed in Section 3.1, this is a generalization of the future mechanism, in that all future values generated by the same object are stored in the same object.[2]

A *get* event reflects the transmission of a future value and has the form:

$$get(u, o, r)$$

where o is the object requesting the future value (through an asynchronous blocking call or *get* statement). We include the future value r in the event even though it is redundant given the corresponding *put* event. This allows additional safety control.

For any interface or class I with methods m_i (for $i \in \{1, 2, , ...\}$), we define the corresponding type $Call[I]$ by one constructor function m_i for

[2]The *comp* event includes redundant information, in the sense that $callee, m(\overline{e})$ are given by the call event with the same future identity. Removing this redundant information gives a more compact representation. However, the redundant specification is useful for specification purposes because a completion event and the corresponding call event are visible through different interfaces of objects other than history objects, as explained in the next section. A solution could be to omit redundant information in the stored transaction history (*trans*), and have a function that enriches the history with the redundant information when needed. A similar discussion applies to *get* events, where the r argument is redundant when the corresponding completion event is present.

```
type Call[I] = ... | m_i: .. *T_k* .. | ... // encoding calls to methods of I/cointerf.
type Comp[I] = ... | m_i | ...     // encoding the corresponding call completions
type Transaction[I] = call: Fid*Any*Any*Call[I]
                    | comp: Fid*Any*Any*Call[I]*Comp[I]
                    | get: Fid*Any*Data
```

Figure 7: Predefined types for transactions

each method m_i, and with input types as given by the parameters of method m_i, and result type as given by the return type of method m_i, see Figure 7. In case a method of the interface has a cointerface, we also add constructor functions for each method of the cointerface. For a class C, then $Call[C]$ includes constructor functions for all local methods, exported methods and cointerface methods. The type $Transaction[I]$ is the union of such calls and completions, adding implicit parameters. For calls the implicit parameters are the future identity, the caller and the callee. For completions the call identity suffices. Here T_k is the type of the kth parameter of method m_i, and the different constructor functions are separated by a bar | (disjoint union).

For interface *Auction* we have that the type $Call[Auction]$ consists of:

```
highest: | open: | close: | makeBid: Nat,     for methods of Auction, plus
newAuction: | closed: | newBid: Nat | youwon: Nat | winner: Bidder,
```

The latter ones are due to the cointerface *Bidder*. The events defined for *Auction* objects, defined by type $Transaction[Auction]$, consist of the following call events made by *bidder* objects (left column) or by the *auction* object (right column):

$call(u, bidder, auction, open())$,	$call(u, bidder, auction, newAuction())$
$call(u, bidder, auction, close())$,	$call(u, auction, bidder, closed())$
$call(u, bidder, auction, makeBid(n))$,	$call(u, auction, bidder, newBid(n))$
$call(u, o, auction, highest())$,	$call(u, auction, bidder, winner(b))$
	$call(u, auction, bidder, youwon(n))$

In this case the get events have the form $get(u, o, r)$, where r is a boolean, natural, or void, and the completion events have the form $comp(u, auction, bidder, youwon(n), void)$ and similar for the other methods.

Notation on histories and transactions. We use dot-notation to extract the different components of a transaction or event. For instance, the caller of

a transaction t is given by $t.caller$. Similarly, we write $t.callee$, $t.fid$, and $t.result$. This syntax is lifted to transaction lists. For instance, the set of callers in a transaction list *trans* is given by $trans.caller$.

The projection operator "/" is defined for lists such that a list projected by a set gives the sublist containing all elements in the set. In particular, $trans/\{.fid = f\}$ is the sublist of transactions where the future identity is f, and $trans/\{.caller = o\}$ is the sublist of transactions where o is the caller.

Furthermore, $trans/Call$ and $trans/Comp$ give the sublists of invocation and completion transactions, respectively. The notation $last(trans/Comp/\{.fid = f\}).result$ means that we take the sublist of completions in *trans* that have future identity equal to f, and then take the value of the last transaction in the sublist. For an interface I we let the projection $trans/I$ denote the subsequence of *trans* restricted to call and completion transactions of methods in the interface I. For instance, we can state that a subclass satisfies the invariant R of an interface I by requiring $R@I(trans/I)$, thereby considering the relevant part of the transaction history.

5. The Implementation of History Objects

Using the above language and notion of histories, we can describe how history objects are implemented. The history object of a contract of class C contains a (private) event history called trans:

```
List[Transaction[C]] trans := nil // restricted by read-only access
```

which is updated by the underlying runtime system by appending each new message from or to the contract object, see Figure 8. This list variable is updated by the runtime system through a predefined put method:

```
Void put(Transaction[C] t) { trans :++ t } // appending t to trans
```

The *put* method is not available to the programmer. The runtime system records every event t (either a call, completion, or get event) in the transaction history by making the call:

$$\mathsf{this}.history.put(t)$$

Note that local synchronous calls, which neither have a future identity nor generate a future, are not recorded in *trans*. Abnormal method termination results in a *comp* event with result **error**.

```
class PREHISTORY[I, C]
implements PreHistory[I] {
// I is the interface of the contract object
// C is the class of the contract object
  type Trans = Transaction[I] // transactions of interface I
  type AllTrans = Transaction[C] // all transactions of the contract class C
  type Hist = List[Trans] // the history seen through the I interface
  type FullHist = List[AllTrans] // the full history, including local events

  FullHist trans := nil // the history, restricted by read-only access

// Void put(AllTrans t){trans :++ t}// used by the runtime system

  T get(Fut[T] f) { // to get a future value when resolved
    when (trans/Comp/{.fid=f})≠nil; // enabled when resolved
     Fut[T] fvalue = last(trans/Comp/{.fid=f}).result;
     return fvalue;
     this.put(get(f,caller,fvalue)) // record the get event
  }

class HISTORY[I, C] implements History[I]
extends PREHISTORY[I,C] { // inherits the parameter "I ctr"
  Trans lastTrans() {return last(trans/Trans)} // error when no last Trans
  Hist getTrans() {return trans/Trans} // returns all visible transactions
  FullHist getAllTrans() {return trans} // returns all transactions
  Hist transOf(Any o) {return trans/{.caller=o} } // all transactions of o
  ...
}
```

Figure 8: Predefined class implementations of PreHistory and History objects

Similarly, a history object includes a public *get* method for each method result of type T which have the following structure:

```
T get(Fut[T] f){when f is resolved ; return the future value; update trans }
```

In order to check that f is resolved, we project the transaction history taking the completions events with f as future identity, and then checking if this

sublist is empty. If not, it will have exactly one element since all future values are unique, and we can extract the future value by *.result*.

A *get* statement is therefore possible in our setting as a blocking call to *get* on the appropriate history object, or as a non-blocking call to *get* using a guard on the corresponding future. Class HISTORY adds definition of methods for extracting the whole or parts of the history, again by means of projection, see explanations in Figure 8.

As mentioned, a history object is generated when a contract object is generated. We therefore use a special notation for contract creation, using the syntax $v :=$ **new** *contractclass* & *historyclass* where *contractclass* gives the class of the contract and *historyclass* gives the class of the associated history object. We let the parameter *ctr* of the history object be bound to the new contract object, and let *ctr.history* be bound to the new history object. We require that the new contract respects (the contract part of) the invariant specification of the history class, as explained in Section 6. The *historyclass* must be given by means of a class adaptation, as in

$v :=$ **new** *SAFEAUCTION* & *AUCTIONHIST*
$v :=$ **new** *AUCTION* & *AUCTIONPRIVATEHIST*
$v :=$ **new** *AUCTION* & *SECUREAUCTIONHIST*
$v :=$ **new** *AUCTION* & *AUCTIONWITHTRANSFER*

(using the adaptations defined in Figures 10, 12, 15, and 17). Type checking requires that the given *contractclass* should be the same as the one used in the adaptation, and that the type of v should be the same (or larger) as the one used in the adaptation. The type of *ctr.history* is the smallest interface of the new history object. We assume that an adaptation has a smallest interface. (An adaptation may still implement several interfaces, say both *SafeHistory* and *AssetHistory*, by introducing an interface extending both.)

Discussion

We observe that the transaction history of an active object, as given by its history object, is sufficient to define the state of the object at the end of a method execution. The state of an active object at its last method completion can be reconstructed from the transaction history, and also the pre-state and post-state of a method with a given future identity. In particular, we may determine the pre-state of a method execution resulting in error. Note that suspension in the middle of a method would require more events to be recorded in the *trans* variables.

Theorem 1 (state recovery)**.** *Given the transaction history (trans) of a contract, the pre- and post-state of a method execution with future identity u can be derived from trans, provided there is a completion with identity u in trans.*

Proof. From the operational semantics of our language, it is clear that each statement is locally deterministic with respect to the object state, apart from *get* statements and the execution start of asynchronously called methods, while local synchronous calls have a deterministic behavior. This means that each method execution is deterministic relative to the prestate, the choice of call message and its content, and the future values observed during the execution. We may assume that all observed future values of a completed method execution are reflected in *trans*. An execution involving a contract object can therefore be seen as a sequence of asynchronously called method executions where each method is executed sequentially without interruptions, apart from local synchronous calls. As local synchronous calls are not reflected in *trans*, the completions of this execution sequence is exactly reflected by *trans* (projected to completions made be the contract object).

The corresponding execution start of such a method execution (with a given call id u) is immediately after the previous completion event of the contract object (and the input parameters are given by the call event in *trans* with this u). Note that this place in *trans* is thereby determined, even though it is not reflected by an event. Thus the sequence of method executions is determined from *trans*, and the post-state of the method execution with identity u is given at the completion state of method execution u, and the pre-state of the method execution with identity u is given at the completion state of the method execution previous to u in *trans*. As mentioned, the value of a **`get`** statement is determined by *trans*, taking the value specified in the corresponding *get* event. Such a *get* event must be present since the completion event is in *trans*. The contract states are therefore calculated from *trans* in a deterministic manner since each method execution start is determined from *trans*. Thus we may recalculate the post- and pre-state of each asynchronous method execution performed by the contract. □

Furthermore, we observe that the history objects define the transaction history in a faithful way due to the write-once/read-many language restriction on the *trans* variable, enforced by the predefined classes, whose fields and methods are "final" and may not be extended by programmers. This gives a protection at the software level somewhat similar to smart contracts/blockchain. If the runtime system also offers protection of unauthorized

write access to these variables by means of write-once/read-many storage or a trusted execution environment, one does not need blockchain technology to guarantee write-once/multiple-read access. If not, one may use blockchains.

6. Contract Specifications and Safety

Our specification language gives rise to executable invariants. This may be exploited in dynamic checking to ensure that there is no contract violation. This gives an extra level of safety and trust between the contract object and its users. This is of course valuable if the contract class has not been formally verified against the contract specification, and even if the contract has been successfully verified, a user may not know if the code was changed after verification time, for instance by means of dynamic class upgrades [30], allowing a class to be replaced by a new version of the class, which means that the class in question (not a history class) is replaced at runtime, updating both existing and future objects of the class.

As explained in Section 4.1, an interface does not have state variables so there is no obvious way to express interface invariants. As mentioned, we therefore use the local transaction history h to express interface invariants, together with types and functions defined over the history. However, the notion of local transactions depends on the particular interface, reflecting what is visible through the interface. In particular, only events related to methods exported through the interface I are visible in I. Thus the set of visible events of a subinterface of I will be larger if additional methods are defined. Similarly, the set of visible events of a class are in general larger than those of an interface of the class, since the class may contain methods not exported through the interface.

We have so far looked at specifications of interfaces and classes defining history objects and contracts. Specifications of history objects represent subsystem specifications for all parties interacting with a given contract, whereas a specification of a contract or contract user restricts the behavior of that object, and is expressed by restrictions on the local history h of the visible events of that object. The visible events of an object o are the *comp* events made by that object, the *call* events made by that object, and the *get* events observed by that object. We let $trans/o$ denote the sub-sequence of *trans* to the visible events of o. In an interface we limit ourselves to the method exported through that interface, and in a class we consider all methods called asynchronously. We let $trans/I$ denote the sub-sequence of

trans to the events of I. This means that an interface I of a contract user or contract o satisfies the specification $R(trans)$ of the associated history object, if $R(trans/o/I)$. Moreover the local history h of a contract user or contract o in an interface I is given by

$$h = trans/o/I$$

For the auction example, the history object invariant may give restrictions on *Bidder* objects, for instance saying that each bidder makes strictly increasing values of bids, as well as on the *Auction* contract, for instance controlling the "youwon" messages. The first restriction may be verified for bidders, but not for the contract, and the second restriction may be verified for the contract, but not for the bidders (since projection reduces it to true).

6.1. Implementation of SafeHistory

An implementation of history objects with built-in safety check is shown in Figure 9, by defining a new version of class *HISTORY* implementing *SafeHistory* of Figure 3. Class *SAFEHISTORY* checks the contract invariant $safe(trans/ctr/I)$ for each new transaction being recorded, by redefining the *put* method so that it checks all output from the contract as well as input to the contract against the contract specification. The initial return void statement ensures that the caller may continue without unnecessary delay while the rest of the *put* method is finishing. For a transaction t violating the safety requirement, we let the "erroneous" transaction, $error(t)$ be stored in the transaction history, where $error : Trans \to Trans$ is a constructor function allowing us to mark transactions as unsafe, and we add profile $error : Comp[I] \to Comp[I]$ etc. so that projection still works. This means that all future values resulting from unsafe method completions will be marked as erroneous. It entails that contract users will observe that output from the contract is not safe when they use the *get* operation.

In order to handle safety violations caused by contract users, we let the *Contract* interface provide a method *reportviolation* that can be used to inform a contract about problems with a contract user, namely that the user does not respect the contract specification. This means that if certain user requirements are not checked by the contract itself, it is detected in the history object and the contract is notified. The added safety check is made by redefining the *put* method. Since the *put* method is not available to the programmer, we require that such redefinitions of *put* are done at a properly authorized level before added to the library of classes for history objects.

```
class SAFEHISTORY[I,C] // inherits the class parameter ctr
  implements SafeHistory[I]
  extends HISTORY[I,C] {
    Void put(Trans t) { // redefined to check contract violations
      return void; // early return to avoid waiting on the caller side
      if safe((trans++t)/ctr/I) then trans:++ t // all fine
      else ctr!reportviolation(t.caller);// notifying the contract asynchronously
           trans:++ error(t) fi} // marking the event as unsafe
}
```

Figure 9: A history class implementation with safety check

A note on efficiency. The use of functions defined over the transaction history may seem inefficient, especially when the history grows in size. However, there are ways to make this more efficient. A history function g inductively defined by equations of the form $g(empty) = init$ and $g(h;t) = rhs(g(h),t)$, can be implemented by a variable g, initialized to *init* and updated by $g := rhs(g,t)$. When all history functions are treated this way, one avoids walking down the entire history whenever the invariant is checked.

A note on detection of violations. We have seen that an unsafe completion event is detected before it is stored in the history object, and the stored value is then marked as erroneous. This means that objects receiving the value (using *get*) will be aware of the contract violation. For call events this is not the case. An unsafe call event is detected after the call has been made, and even though the contract is notified, it may be too late. To make an improvement here, one could let the history object behave as a wrapper around the contract, letting all transactions between a contract and its users, pass through the wrapper and be checked before coming through, using the general wrapper concept of [37]. The disadvantage of this is that nontrivial checks will slow down the overall performance. If a history object invariant restricts the completion events, which is the natural place to make requirements, our approach with separate history objects working independently seems to be an appropriate solution.

6.2. Specification of the Auction Example

We reconsider the auction example, using the interface *ActionHist* and class *SAFEAUCTION* given in Figure 10. Note that the functions defined

```
interface AuctionHist extends SafeHistory[Auction]{
  type Trans = Transaction[Auction] // abbreviated data type definition
  // type Hist = List[Trans]; inherited
  // Functions defined (inductively) over the transaction history
  func bid: Hist → Nat // calculates the highest bid from trans
  func bidder: Hist → Bidder // calculates the highest bidder
  func safe: Hist → Bool // ensures that the winner is the real winner
  var Hist q, Nat n, Bidder b
  bid(nil) = 0
  bid(q; comp(_,_,ctr,makeBid(n),true)) = max(n,bid(q))
  bid(q; comp(_,_,ctr,close(),true)) = 0 // auction closed, high bid reset
  bid(q; others) = bid(q)

  bidder(nil) = null
  bidder(q; comp(_,b,ctr,makeBid(n),true)) =
                                    if n>bid(q) then b else bidder(q)
  // auction closed, highbidder reset:
  bidder(q; comp(_,_,ctr,close(),true)) = null
  bidder(q; others) = bidder(q)

  safe(nil) = true
  safe(q; call(_,ctr,b, youwon(n))) = (b=bidder(q)) and (n=bid(q))
  safe(q; others) = true
  invar safe(h) // Finally, the contract specification
}
```

```
class SAFEAUCTION extends AUCTION {//the contract implementation
  // Note that h = trans here since trans = trans/this/SAFEAUCTION
  invar safe@AuctionHist(h) and highBid=bid(h) and highBidder=bidder(h)
}

class AUCTIONHIST(Auction a) implements AuctionHist
  adapts SAFEHISTORY[Auction, SAFEAUCTION] {
}
```

Figure 10: The Safe Auction class and the related history object interface and class

in an interface are available in any class supporting the interface, and may occur in executable code. The local history h of *AuctionHist* is given by $trans/AuctionHist$, which simply is $trans$ in this case since there all methods of the class implementation are visible through *AuctionHist*.

The contract specification expressing the main property of the *Auction* says that $safe(h)$ holds, as well as $o = bidder(h)$ and $n = bid(h)$ when $o!youwon(n)$ is called (for some o). This ensures that the *youwon* call is sent to the right actor, i.e., the one winning the auction according to the transaction history, and that the winning amount is the correct highest bid according to the history. Note that both *bid* and *bidder* are calculated from successful *makeBid* transactions, i.e., with return value *true*. As shown in Section 7, this property can be verified for the class implementation of *AuctionHist* given in Figure 10.

7. Verification

In this section, we discuss how to verify contract specifications by a Hoare-style logic for partial correctness (i.e., assuming termination of statements). We show how to verify a class invariant in a class-wise manner and demonstrate the verification technique on the auction example. The invariant R of a class C may talk about fields, class parameters, and its local history h. We need not consider events reflecting start of execution of a method in the class since these can be derived from the history. The reasoning system here is simpler than in [15, 35, 36], since we may consider a smaller event set for method interaction due to our notion of initial guards. With the presence of futures, we have a more expressive language than in [14]. Reasoning about an object o taking part in a smart contract can only verify the role of that object. The local history of o of class C reflecting this role is the transaction history of the associated history object restricted to the events generated by o, i.e.,

$$h = trans/o/C$$

To verify the whole contract, as stated in the history object, we need in general to combine the invariants of the objects taking part in the contract, using a rule for composition. The objects engaging in a contract can be seen as a subsystem of concurrent objects, and we may use the composition rule of [35, 36]. Adapted to our case, the composition rule generates an invariant $Inv(trans)$ of the history object (where $trans$ is the history of all events seen

by the history object) by forming the conjunction of all the local invariants $R^{\,h,\ \mathsf{this}}_{trans/o/C,\ o}$ of each object o in the subsystem together with a wellformedness predicate stating that each call transaction has a unique future identity, and that a call comes before the corresponding completion transaction (the one with the same future identity), and that a result read (in a *get* event) is the same as the generated result (in the preceding completion event with the same future identity). The notation R^v_e denotes the substitution of (free) occurrences of v in R by e, and $R^{\overline{v}}_{\overline{e}}$ simultaneous substitutions. The replacement of this by o is needed to formalize the fact that what is called this in the class invariant is the object o in the contract.

To verify that R is an invariant of a given class C, we must prove that the invariant is satisfied initially, i.e., that R holds for an empty history and initial field values, and that each method of the class maintains R. The verification of a method inside a class is done by sequential Hoare-style reasoning. The Hoare triple $[P]\ s\ [Q]$ expresses that if the predicate P holds in the pre-state of s then the predicate Q holds in the post-state, provided s terminates normally. If Q implies P, then Q is said to be an invariant of s.

To prove the invariance of R for a method m with parameters $\overline{x}$ and method body **when** guard; s; **return** e, we need to verify:

$$[R \wedge guard] \quad s \quad [R]$$

This triple can be verified locally in the class by ordinary sequential Hoare logic. In particular, the Hoare axiom for assignment is given by:

$$[Q^v_e]\ v := e\ [Q]$$

This axiom holds since there is no remote field access (and therefore no semantical aliasing problems) and since expressions are pure. Several other statements may be reduced to assignments: The special syntax $h :{+}{+}\ t$ is semantically the same as the assignment $h := (h; t)$. The return statement **return** e is semantically equivalent to the assignment

$$h :{+}{+}\ comp(\mathsf{fid}, \mathsf{caller}, \mathsf{this}, m(\overline{x}), e)$$

As mentioned, we assume that there is exactly one return statement in each branch of a method body, and for simplicity we assume that method parameters are read-only.

Furthermore, an asynchronous call $o!m(\overline{e})$ is treated as the two assignments $f :=$ **new Fut**; $h :{+}{+} call(f, this, o, m(\overline{e}))$ where the first assignment

represents a non-deterministic assignment to f resulting in a fresh future identity (like an object creation statement). The Hoare rule for this assignment is $[\forall f\,.\,f \notin h \Rightarrow Q]\ f := \texttt{new Fut}\ [Q]$ where the universal quantifier corresponds to non-determinism, and the condition ensures freshness of f, i.e., that f has not already occurred in a transaction in h. Thus, we derive the following rule for asynchronous calls:

$$[\forall f\,.\,f \notin h \Rightarrow Q^{h}_{(h;call(f,this,o,m(\overline{e})))}]\quad o!m(\overline{e})\quad [Q]$$

The rule for the *get* statement is given by:

$$[\forall v\,.\,Q^{\,h}_{h;get(f,this,v)})]\quad v := \texttt{get}\ f\quad [Q]$$

where the universal quantifier on v' corresponds to a non-deterministic assignment to v, which reflects that the read value is locally unknown. In the compositional rule, this value is resolved through wellformedness of the contract history (using the corresponding completion event and the wellformedness predicate).

Combining these two rules, we obtain a similar rule for non-local asynchronous blocking call ($o \neq this$):

$$[\forall f, v'.f \notin h \wedge o \neq \mathsf{this} \Rightarrow Q^{\,v,\ h}_{v',(h;call(f,this,o,m(\overline{e}));comp(f,this,o,m(\overline{e}),v'))}]v := o.m(\overline{e})\ [Q]$$

since $v := o.m(\overline{e})$ is semantically the same as the statement sequence $f := \texttt{new Fut};\ h :{+}{+}\ call(f, this, o, m(\overline{e})); v := \texttt{get}\ f$ when the call is non-local. The prime on v' is needed in case v occurs in $\overline{e}$ (which is the old v and should not be quantified). (Local calls can be treated as in standard sequential reasoning.)

In the setting of partial correctness, $[P]\,s\,[Q]$ assumes normal termination of s (and no errors). Thus it does not ensure error-free execution. However, reasoning about errors is needed in the presence of error handlers, because a handler may turn an error into a normal value and then the assumption of normal termination is satisfied. Reasoning about error handlers can be done as indicated below for handlers on assignments and calls.

One may treat an assignment with a handler, $v := e < s >$, as the assignment $v := e$ if e does not result in an error, otherwise s. To distinguish the two cases, one may use a predicate WD expressing welldefinedness, letting $WD(e)$ be true if the evaluation of e results in a normal value and false if it results in an error. (This could be expressed as $(e = e)\#false$ extending the

error handling to the functional level, using the syntax $e\#e'$.) For instance, $WD(last(q))$ is $(q \neq nil)$, which can be obtained from the definition of *last* in Figure 5. The rule is then:

$$\frac{[P]\ s\ [Q]}{[\mathbf{if}\ WD(e)\ \mathbf{then}\ Q^{\,v}_{\,e}\ \mathbf{else}\ P]\ v := e < s >\ [Q]}$$

Reasoning about an asynchronous blocking call with a handler, $v := o.m(\overline{e}) < s >$, can be done by the rule:

$$\frac{[P]\ s\ [Q]}{[\forall f, v' \,.\, f \notin h \wedge o \neq \mathsf{this} \Rightarrow Q' \vee P']\ v := o.m(\overline{e}) < s >\ [Q]}$$

where Q' is Q with the substitutions given in the rule for asynchronous blocking call, and P' is $P^{\,h}_{h;\,call(f,this,o,m(\overline{e}));\,comp(f,this,o,m(\overline{e}),error)}$. The two extensions of the history in the precondition are disjoint since quantified variables range over defined values. A key point here is that the handler can be connected to a particular transaction in the history.

One could consider other forms of error handlers (for instance at the end of a method body) in a similar manner, and reasoning about raising exceptions can be done as usual. We do not discuss reasoning about roll-backs, which can be non-trivial, especially if it is not clear at verification time how far the roll-back should go.

7.1. Verification of the Auction Example

Verification of the Class Invariant

We show how to verify that the implementation of the auction system given in Figure 10 (by class *SAFEAUCTION*) satisfies its invariant. For our example, it suffices to consider completion events since only these are used in the contract specification in *AuctionHist*.

We prove that class *SAFEAUCTION* satisfies its invariant R, namely:

$$safe(h) \wedge highBid = bid(h) \wedge highBidder = bidder(h)$$

where *safe* is defined in interface *AuctionHist*. The invariant must hold initially and be maintained by each method (except non-visible methods called synchronously). Initially the history is empty, *highBid* is 0, and *highBidder* is *null* (the initial value of object references). Thus we need to prove $safe(nil) \wedge 0 = bidder(nil) \wedge null = bidder(nil)$, which is trivial.

Method *highest*() does not change any fields, and the completion of *highest* has no effect on the invariant, so verification is also trivial (since R implies R).

For method *open* we need to verify:

$$[R \wedge \ \mathbf{not}\ isopen] \quad owner := caller; isopen := true \quad [R^{\,h}_{h;t}]$$

where t is $comp(\mathsf{fid}, \mathsf{caller}, \mathsf{this}, open(), void)$. This gives the following verification condition:

$$R \wedge \ \mathbf{not}\ isopen \Rightarrow R^{\,h,owner,isopen}_{(h;t),caller,true}$$

with t as above. It is straightforward.

For method *makeBid* we need to verify the following two triples:

$$[R \wedge not(open \wedge x > highBid)] \quad bidders : +\!+ caller \quad [R^{\,h}_{h;t}]$$

where t is $comp(\mathsf{fid}, \mathsf{caller}, \mathsf{this}, makeBid(x), false)$, and:

$$[R \wedge open \wedge x{>}highBid]\ highBid := x; h : +\!+ t'; highBidder := caller\ [R^{\,h}_{h;t}]$$

where t is $comp(\mathsf{fid}, \mathsf{caller}, \mathsf{this}, makeBid(x), true)$, and replacing the asynchronous call by h:++ t' where t' is the call event reflecting the newBid call, which does not influence the invariant. We here ignore the assignment bidders:++caller since the variable bidders does not occur in the invariant. The first triple is trivial since $R^{\,h}_{h;t}$ reduces to R in this case. The second verification condition reduces to:

$$R \wedge open \wedge x{>}highBid \Rightarrow R^{\,h,highBidder,highBid}_{(h;t';t),caller,x}$$

by sequential Hoare analysis. This gives the verification condition:

$$x = bid(h; t'; t) \wedge caller = bidder(h; t'; t)$$

under assumption of:

$$safe(h) \wedge highBid = bid(h) \wedge highBidder = bidder(h) \wedge open \wedge x{>}highBid$$

This reduces to true by the function definitions of *bid*, *bidder*, and *safe*.

Finally, we consider method *close*. Ignoring assignments and calls not affecting the verification, the most challenging branch reduces to the triple:

$[R \wedge caller{=}owner \wedge highBid{>}0]$
$highBidder!youwon(highBid); highBidder := null \ [R^{\,h}_{h;t}]$

which gives the verification condition:

$$R \wedge caller{=}owner \wedge highBid{>}0 \Rightarrow (\forall f \,.\, f \notin h \Rightarrow \forall f \,.\, R^{\,h,\ highBidder}_{(h;t';t),\ null})$$

where t is the *close* completion $comp(\mathsf{fid}, \mathsf{caller}, \mathsf{this}, close(), true)$, and t' is $call(f, \mathsf{this}, highBidder, youwon(highBid))$ reflecting the *youwon* call. This verification condition follows from the definitions of the invariant and the *safe*, *bid*, and *bidder* functions.

This completes the proof of invariance of the invariant.

Verification of the Contract Invariant

By the rule for composition we may conclude $safe(trans)$, which ensures the contract specification of the history object. The fact that the contract class invariant alone ensures the contract specification reflects that the contract specification does not entail any restrictions on the contract users. However, imagine that the auction history invariant did state that the current owner may not make a bid, say:

$$safe(h; call(\_, b, ctr, makeBid(n))) = (b \neq curowner(h))$$

with *curowner* defined similarly as the other history functions:

```
func curowner: Hist → Bidder
curowner(nil) = null
curowner(q; comp(_,b,_,open(),_)) = b
curowner(q; comp(_,b,_,close(),true)) = null
curowner(q; others) = curowner(q)
```

Then this requirement would have to be verified for *Bidder* objects, in order to conclude that $safe(trans)$ holds for the auction contract and its users.

8. Adding Privacy Aspects

A main privacy concern is the handling of information about a particular data subject, such as an identifiable natural person. At the abstraction level of our modeling language, we may capture a data subject by an object, more specifically, by an object supporting a predefined interface *Subject*

```
class PRIVATEHISTORY[I, C] implements PrivateHistory[I]
extends HISTORY[I,C] {
// redefines all methods by using restrict:
func restrict: Any * Any -> Any
func restrict: Any * Data -> Data
func restrict: Any * Hist -> Hist
var Any o, Any o', Data d, Transaction t // variables for function definitions
restrict(o, o' ) = restrict(o, const) = const // for each constant constructor
restrict(o, c(.. ,d, .. )) = if trusted(o) then c(.. ,d, .. )
                  else c(.. ,restrict(o, d), ..) // for each constructor function c
                  // including call, comp, get for the Transaction type
  // For the list type this means:
  restrict(o, nil) = nil
  restrict(o, h++t) = if trusted(o) then h++t
                      else restrict(o, h)++restrict(o, t)
  // Redefinition of methods using restrict:
  T get(Fut[T] f) { // to get a future value when resolved
    when (trans/Comp/{.fid=f})≠nil;
     Fut[T] fvalue = restrict(caller, last(trans/Comp/{.fid=f}).result);
     return fvalue;
     this.put(get(f,caller,fvalue)) } // record the get event

  Trans lastTrans() {return restrict(caller, last(trans/Trans))}
  Hist getTrans() {return restrict(caller, trans/Trans)}
  FullHist getAllTrans() {return restrict(caller, trans)}
  Hist transOf(Any o) {return restrict(caller, trans/{.caller=o}) }
  ...
}
```

Figure 11: Predefined implementation of privacy-restricted history objects

(*Subject* < *Any*). For instance, a contract party will typically be a data subject, but not the contracts themselves nor the history objects since they do not reflect a natural entity. In general we may want to avoid the spreading of information about data subjects, so-called personal data, except to the subject itself. Such spreading is manifested through parameters of method calls or result values of completions and get events. One way of enabling privacy

```
interface DefaultPrivateHistory[I] extends PrivateHistory[I] {
var Any o
trusted(o) = false // no one is trusted
}

interface AuctionPrivateHistory extends PrivateHistory[Auction] {
var Any o
// bidders that have made successful bids are trusted:
trusted(o) = comp(_, o, ctr, makeBid(n), true) ∈ h
}

class DEFAULTPRIVATEHIST[I, C] implements DefaultPrivateHistory[I]
adapts PRIVATEHISTORY[I,C] { // trusted is now defined (by false)
}

class AUCTIONPRIVATEHIST implements AuctionPrivateHistory
adapts PRIVATEHISTORY[Auction, AUCTION] {
// trusted is as defined in AuctionPrivateHistory
}
```

Figure 12: Alternative definitions of *trusted* and examples of usage

control is to limit the information about subjects through these events.

As default, an object implementing an interface I is allowing output of information about itself in interaction with other subjects through the methods of I, and when using an interface J the object is assumed to allow information about itself submitted through calls to subjects of interface J. But we would like to restrict the spreading of information about other subjects through history objects.

In the auction example, interface *Bidder* is potentially spreading information about a subject (the winner) through method *winner*. This problem could be detected statically. (In the auction implementation this method is used to inform the owner about the winner, something which is not really needed and the method could be removed.) No other spreading of personal data is possible with the *Bidder* and *Auction* interfaces. However, spreading of personal data through the history objects is possible since anyone may access (parts of) the transaction history.

In order to solve this problem, we define a subclass of *HISTORY* that restricts the information in each transaction. It is not a good solution to remove this information from the history objects since it is essential for security and safety checking. But, we can filter out personal information in the information going from history objects to other objects. We allow the sending of information about an object to itself since self access should be supported, and we allow information about contract transactions to be sent to trusted objects using the *trusted* function specified by the contract designer. All other references to a subject S are replaced by a special symbol representing hidden information, for this purpose we use the $*$ symbol (of type *Subject*). Figure 11 shows how to implement this by means of a class *PRIVATEHISTORY* implementing *PrivateHistory*. The implementation is using a function *restrict* to filter away private information send to an object o. For a history h, $restrict(o, h)$ filters away all occurrences of subject references other than o, unless o is trusted in which case nothing is filtered away.

For strict privacy control, one may define $trusted(o) = false$, saying that no parties can be trusted. This is shown in interface *DefaultPrivateHistory* of Figure 12, and class *DEFAULTPRIVATEHIST* is used to instantiate history objects with this privacy policy. Note that the interface must extend *PrivateHistory* in order to be able to talk about the *trusted* function, and the class must extend *PRIVATEHISTORY* in order to connect the definition of *trusted* to the *restrict* function. If this policy is used for an auction contract a and the transaction history ends with:

$$call(f1, o1, a, makebid(10)); call(f2, o2, a, makebid(20));$$
$$comp(f2, o2, a, makebid(20), true); comp(f1, o1, a, makebid(10), false)$$

the restricted history when communicated to object $o1$ ends with:

$$call(f1, o1, a, makebid(10)); call(f2, *, a, makebid(20));$$
$$comp(f2, *, a, makebid(20), true); comp(f1, o1, a, makebid(10), false).$$

where $*$ indicates hiding of private information as explained.

For the auction example, we may give a more lenient privacy policy. If we trust all actual bidders, we may instead specify that all successful bidders are trusted. This is done in interface *AuctionPrivateHistory* of Figure 12, and the corresponding class to be used for this policy is given by *AUCTIONPRIVATEHIST*. As an alternative, if we want to restrict trusted

bidders to successful bidders of the current auction session (started with a successful *open*), we could define *trusted*(o) by:

$$comp(\_, o, ctr, makeBid(n), true) \in (h \textbf{ after } comp(\_, \_, ctr, open(), true))$$

where h **after** s takes the part of h after the last event in the event set s. In either case, the transaction example above gives hiding of information relative to $o1$ as before, since $o1$ is not trusted; but if the four transactions are sent to $o2$, there would be no hiding since $o2$ is trusted.

9. Adding Security Aspects

```
// Subclass with build-in security
class SECUREHIST[I,C] implements SecureHistory[I]
extends HISTORY[I,C]{

  T get(Fut[T] f) {return if blacklisted(caller) then error
                    else get@HISTORY(f) fi }
  // similar for the other retrieval methods ...
}
```

Figure 13: A predefined class for history objects using blacklisting to control security

```
interface DefaultSecureHistory[I] extends SecureHistory[I] {
  var Any o
  // parties that have produced erroneous return values are blacklisted:
  blacklisted(h,o) = not WD(h/o/Comp)
}
```

```
class DEFAULTSECUREHIST[I, C] implements DefaultSecureHistory[I]
adapts SECUREHIST[I,C] {
  // inherits definition of blacklisted from DefaultSecureHistory
}
```

Figure 14: A user-adapted definition of history objects using blacklisting to control security

```
interface SecureAuctionHistory extends SecureHistory[Auction] {
  var Any o
  // parties that have produced improper close calls are blacklisted:
  blacklisted(h,o) = not (h/Comp(_,o,ctr,close(_),false) = nil)
}
```

```
class SECUREAUCTIONHIST implements SecureAuctionHistory
adapts SECUREHIST[Auction, AUCTION] {
  // inherits definition of blacklisted from SecureAuctionHistory
}
```

Figure 15: Adapted auction history objects with improved security

A security addition can be defined by implementing class *SecureHistory* of Figure 3 and providing a definition of *blacklisted* reflecting contract parties that have violated some property. We show a class implementation in Figure 13 where blacklisted objects are not obtaining any information from the history object. The *get* method (and similarly the other methods) is redefined such that blacklisted callers cannot get any information. Figure 14 includes a general definition of the *blacklisted* function given by interface *DefaultSecureHist*. Here contract parties that have caused errors are blacklisted, since attackers often do not behave properly, using the *WD* function to check for errors. The interface and class defined in Figure 14 can be user-defined. As an alternative, one could reuse the concept of *trusted* and define:

$$blacklisted(h, o) = \textbf{not}\ trusted(o)\ \textbf{or not}\ WD(h/o/Comp)$$

The definition of who is blacklisted can be further redefined to accommodate more sophisticated checking.

Similarly, we can handle security by restricting access to information in the smart contracts for specific parties. We show a specialization for the Auction example, where an interface *SecureAuctionHistory* specifies that objects doing improper *close* calls are blacklisted. Class *SECUREAUCTIONHIST* instantiates class *SECUREHIST* for this specification of *blacklisted*.

Blacklisting could also be connected to failure of transfer of assets, which is considered next. One could then look at the erroneous completion of *transfer* calls and blacklist objects that caused such calls.

10. Adding Transfer of Assets

A smart contract is often combined with the aspect of transfer of assets. Transfer of assets can be associated with each transaction. The cost of a transaction depends on the kind and parameters of the transaction, and possibly the local history h. Our framework provides a cost function to be specified by contract designers, which may capture contract dependent costs of selling or buying services. This function should be defined in subinterfaces of *AssetHistory*, specifying who carries the cost, who benefits from the cost, as well as the number of "cost units", given by a natural number. We use the type Transfer to capture such costs:

$$\textbf{type}\ \mathit{Transfer} = \mathit{none} : \mid (\mathit{Any} * \mathit{Any} * \mathit{Nat})$$
$$\textbf{func}\ \mathit{cost} : \mathit{Transition} \to \mathit{Transfer}$$

A non-empty transfer is given by a triple of form $(from, to, amount)$ where *from* defines the paying party, *to* defines the beneficiary, and *amount* defines the amount transferred, and *none* denotes the empty transfer. The *cost* function is simply a function from a given transaction to such a triple. An asynchronous method call to a contract c from some party p may have the default cost $(p, c, 1)$ saying that each transaction has a small cost for the caller. This is helpful in avoiding distributed denial of service (DDoS) attacks since an attack on the contract by a larger number of calls would impose non-trivial costs, which makes such attacks harder, at least for attackers making these calls. Furthermore, there may be a cost associated with certain transactions. In general, there may be a cost for callers of contract methods, while the contract object may define cost for others depending on the particular method and the history. Static checking may be used to detect DDoS where an attacker directly or indirectly causes call flooding on a victim; and the approach of [17] can easily be adjusted to detect flooding with high cost. The implementation of the auction contract is robust against such attacks.

The handling of assets will be an attractive target for an attacker, either to obtain funds for himself or to harm others. It is essential to protect against such possibilities. A more restrictive cost handling could be to insist that an object may not impose a cost on someone else, i.e., for a call with a cost, the payer must be the caller. One could also consider restricting the payee to be the callee. This could easily be captured by an alternative history interface and predefined class implementation. In this case the cost could

simply be captured as a function from transactions (and the history) to natural numbers, using caller and callee as payer and payee. Note that this more restricted version of transfers would make the auction cost specifications in Figure 17 impossible since a *youwon* call imposes a cost on the callee. Instead we could add prepayments by defining a cost on a (successful) *makeBid*(n) call to the contract (the cost could be n minus earlier prepayments of the caller in the current session), and let the contract pay the winner upon successful *close* completions and return the prepayments made by all bidders except the highest bidder. This can conveniently be formulated by functions over the history.

Figure 16 shows an interface extending *AssetHistory* with functionality for obtaining the accumulated balance of any contract party using functions over histories to calculate this balance. This shows that the balance of each partner with respect to the contract interactions is defined by the history object, and one may therefore use the history object as a digital bank.

We may let the smart contract store the assets, or use a more traditional bank service. In order to make the actual transfer in the latter case, we may then let the contract object use a bank service, here given as a contract object (the *bank* parameter), sending transfer calls as according to the cost function. These transfer calls are made by the contract object according to the predefined class implementation of *AssetHistory*, by redefining the *put* method, as done in Figure 16. We indicate how transfer failures are detected, but not how to deal with such failures, which possibly may involve roll-backs. Details of the bank interface is not given, as this would go beyond the purposes of this paper. Note that the handling of transfers is performed automatically by the history objects according to the predefined implementation of *AssetHistory*.

Figure 17 shows an extension of the auction example with costs. Here all calls to the contract has a small cost (1 unit) to limit DDoS attacks. Furthermore, there is a cost when a new auction is successfully opened since this reflects an essential service of the auction. Finally, the winner of an auction is required to pay the agreed amount to the owner of the auction. This is reflected in the *youwon* method, and the auction contract imposes a cost on the callee (the winner) using the history to identify the beneficiary (the owner). Other methods called by the contract have no cost (i.e., *none*).

As mentioned in Section 4, programmers are not allowed to modify the methods of the predefined classes implementing history objects, and may therefore not redefine the *put* method to their benefits. There should be

```
interface AccAssetHistory[I] extends AssetHistory[I] {
    func costfor: Transfer * Any -> Int
    costfor(none, b) = 0 // defining the cost of a transfer for a given object
    costfor((x,y,n), b) = if x=y then 0 else
                          if x=b then −n else if y=b then n else 0
    func balance : Trans * Any → Int // the accumulated balance of a party
    balance(nil, b) = 0 // defining the balance of an object b
    balance((h;t), b) = balance(h,b) + costfor(t,b)
}

class ASSETHISTORY [I, C] (Bank bank)
  implements AccAssetHistory[I]
  extends HISTORY[I,C] {

    Void put(Trans t) { // redefined to check contract violations
      trans:++ t;
      bank.transfer(cost(t)) < deal with transfer failure >
   }

  Int myBalance() {return balance(h,caller)} // the assets of caller
  Int contractBalance() {return balance(h,ctx)} // the contract's assets %Q10
}
```

Figure 16: A general implementation of AssetHistory by a predefined interface and class

a library of predefined classes covering all combinations of history objects with/without safety control, security control, privacy control, and cost control. These sixteen classes are easily derived from those shown. Our framework allows contract designers to specify cost, but not to implement the handling of cost, since that is given by the predefined classes.

11. Evaluation

11.1. Difference between our Language and Solidity

In general, our language is more high-level and abstract than Solidity [44]. Our language is oriented towards a compositional semantics and class-wise reasoning. In contrast to our approach, Solidity does not support reasoning

```
interface AuctionWithTransfer extends AccAssetHistory[Auction]{
 // inherited: ctr is the current contract, b a bidder, and n a natural number
 cost(call(f,b,ctr,_)) = (b,ctr,1) // all calls to ctr have a small cost
 cost(comp(f,b,ctr,open(),true)) = (b,ctr,1) // true indicates successful open
 cost(call(f,ctr,b, youwon(n))) = (b,owner(h),n) // the winner must pay owner
 cost(others) = none // the other contract calls have no cost
}

class AUCTIONWITHTRANSFER implements AuctionWithTransfer
adapts ASSETHISTORY[Auction, AUCTION] { // cost function is defined
}
```

Figure 17: User-defined specification of auction costs and a class adaptation

about contract specifications by a logic or verification system. Thus the focus of the languages is different, nevertheless we may try to compare the expressiveness.

The communication mechanisms are similar, but are following more closely the object-oriented style in our case. We allow one-way and two-way message passing, as well as multi-, broadcasting and first-class futures. The Solidity notion of *msg.sender* corresponds to *caller* in our language.

Every contract in Solidity can include declarations of State Variables that contain persistent data, similar to class fields in our language, Functions that can modify variables, like methods in our language, Function Modifiers, typically through require statements, comparable to guards in our language, Events, similar to transactions in our setting, Struct Types (record type) and Enum Types, which can be seen as special cases or implementations of user-defined data types in our language. Besides, contracts support encapsulation (visibility attributes) and may inherit from other contracts – in our setting this follows from our notion of interface abstraction and inheritance.

Through fallback functions, custom handling of messages that do not specify a concrete function to call is supported in Solidity. Values are returned from functions by means of return statements/variables. In our setting, there are no fallback functions since all call messages will be understood due to static typing as explained below. A typical queue order in our language is defined by taking the oldest enabled call first (priority calls could be added). The use of guards allows calls to be delayed and thereby control

the scheduling of calls from within the programming language.

Strong typing. Like Solidity, our language supports strong typing of variables and expressions. Calls are strongly typed, and for a call $o.m(\overline{e})$ it is checked that the type of o is an interface that supports a method m such that the actual parameters respect the parameter types of that method, and one can guarantee that the caller o cannot be null by static over-approximation. Due to the cointerface mechanism, the caller has a type, and thus even callbacks of form $caller.m(\overline{e})$ can be strongly typed [31], in contrast to Solidity. The primitives call and delegated in Solidity give rise to untyped calls.

Visibility of attributes. Solidity uses a number of primitives, including private, public, internal, and external to declare visibility of methods/functions from the outside, whereas our language is based on interface abstractions, i.e., we use interfaces to define visible parts of a class. Only methods declared in an interface are visible. It is statically checked that a class defines (either explicitly or implicitly through inheritance) each method declared in an interface implemented by the class. Thus "method not understood" errors are not possible, as explained above, which explains why fallback functions are not needed in our setting.

Reentrance. Reentrancy happens when an object is interrupted during execution, and can be called again before its previous invocations complete execution. Reentrance is a well-known source of undesired behavior in Solidity programs. Reentrance is also possible in our setting, but only in invariant states and when the guard is satisfied. The guard makes it possible to delay calls that would break the invariant, and thereby ensure desired ordering restrictions on the transactions. This mechanism is valuable in avoiding undesired behavior.

Inheritance. Multiple inheritance and polymorphism are supported in Solidity. This is controlled by means of the keywords virtual and override. Using the syntax ContractName.functionName(), and super.functionName(), one can call functions higher up in the hierarchy, and one level up in the flattened inheritance hierarchy, respectively. Our language supports multiple inheritance as well and has a similar way of selecting methods from particular superclasses. For simplicity, we do not discuss overloading here, but in our setting of strong typing, overloading with respect to both method parameters and method result could be done as in [30].

Function modifiers. By using modifiers in Solidity, the behavior of functions can be modified in a declarative way. In particular, a require statement can check a condition prior to executing the function, possibly causing

abnormal termination and roll-back of the state. The require statement can be compared to our notion of guards, which are also checked at the start of a method execution, but do not cause abnormal termination, instead, they may cause the execution of the current method to be delayed until the guard is satisfied. For instance, if the guard is a future check $f?$, this gives a non-blocking way of waiting for a future value. Reasoning about guards is straight forward, as discussed in section 7, whereas reasoning about roll-backs is non-trivial since it depends on dynamic information about which calls have been made. Our setting still supports runtime roll-backs since the state of a contract at the point of a specific call in the past can be calculated from the transaction history in the history object. Invariant reasoning about modifiers can also be challenging, since it is not sufficient that the modified function maintains the invariant.

Flexibility is provided by class adaptations in our framework. A predefined class can be adapted by means of contract dependent specifications.

Assets. Our approach includes a brief discussion on the treatment of transfer of assets. We let the aspect of assets and transfer be defined by the history objects by predefined and protected subclasses, whereas the cost aspects are specified by contract designers in appropriate interfaces. This gives trust at the software level since no programmer can redefine the code of the history objects dealing with transfer.

Predefined data types. Solidity has a large number of predefined numeric types. We have a small number of predefined types and can mimic other types by inductive data type definitions. Struct definitions are composed of a list of pairs of type names and field names. These can be defined in our setting by data type definitions using product and disjoint union.

Expressions. Function calls in Solidity can be of several types: internal, external, delegate, and calls to certain built-in functions. Internal function calls are simply jumps in the code of the current account. External calls cause a message to be sent over the Ethereum network, executing code on another account. Delegate calls exist to provide a functionality akin to shared libraries, by allowing code from another account to directly operate on the storage of the calling account. The semantics of external and delegate calls are notorious sources of bugs in contracts [4], notably the DAO [12] and Parity multi-sig contract [38] incidents. Our language supports a modular semantics, and the mentioned bugs do not apply. As mentioned, we may add a reasoning-friendly delegation mechanism [35].

Error types in Solidity. A strong detection of syntax errors is advan-

tageous since that reduces the amount of runtime errors. Runtime errors appear after the smart contract is deployed to the Ethereum blockchain and the Solidity code has been compiled to a bytecode that can be understood by the Ethereum virtual machine. A runtime error happens when EVM detects something wrong with the smart contract or is making transactions against the logic of the code. All the state changes that are caused by a transaction resulting in error are canceled, and the transaction is reverted, and depending on the kind of error, all the gas of the transaction is consumed or some of it is refunded. Common runtime errors are: out-of-gas error, revert error, invalid opcode error, invalid jump error, stack overflow, and stack underflow. An out-of-gas error happens when there is insufficient gas to complete a transaction. If we try to execute a transaction that is not according to the logic of the smart contract, then EVM returns an error called revert error and the transaction will be reverted. Invalid opcode happens when we try to execute a code that does not exist. Invalid jumps occur when we try to execute a function that does not exist, for instance, if we try to call a function in another smart contract at an address that does not exist. It can also happen if we use assembly in Solidity and point to a wrong location in the code. Stack overflow happens when there are too many recursive calls. In Solidity, the stack can be at most 1024 frames deep, which means a function can only call itself 1024 times. This is a kind of error that is not specific to Solidity and may also happen in many other programming languages. Stack underflow happens when we try to read an item on an empty stack.

The mentioned runtime errors do not occur for our language, but errors may result from execution of partial functions and methods raising errors. This is due to a stronger type system and a simpler and more abstract semantics. For instance, the runtime stack is considered unbounded. However, there may be calls made to object expressions that are null, which again may result in non-terminating **get** statements. This issue can be avoided by static type checking, over-approximating the type of non-null variables and requiring that any callee and actual object parameters are of non-null types. Object generation, this, and caller can be assumed to be of non-null types.

Finally we have logic errors. A logic error reflects a problem in the logic of the smart contract, and differs from a runtime error in that it is not easily detected. Logic errors happen when the developer has made a mistake, or there are open loopholes in a smart contract that can be exploited by attackers. Basically, this kind of error makes a smart contract either dangerous or makes it produce false result. These kinds of bugs are the most serious and

also the hardest to fix because there are no tools that can automatically run through the smart contract and detect logic errors.

Formal verification can be useful in the analysis of logic errors in smart contracts. An example of a logic error is the famous 2016 reentrancy attack of the Distributed Autonomous Organizations (DAO) smart contract, where a code that connects a set of smart contracts together and functions as a governance mechanism, was a result of a logic error. A reentrancy attack can happen when we make an external call to another untrusted contract before it resolves an error. If the untrusted contract is malicious, it can take over the control flow of the original smart contract's code; for instance it may repeatedly withdraw Ether from the smart contract. In contrast, our approach has a modular semantics, which makes it harder to attack a contract provider, and it gives support of behavioral specification by means of contract invariants that can be checked at runtime. Furthermore, our approach supports class-wise verification method, which can guarantee absence of certain logical errors at the cost of interactive theorem proving.

11.2. Difference between our Framework and Blockchain

Blockchains have the advantage of being transparent, immutable, and corruption-free. Our history objects offer similar advantages at the programming language level:

- There is read access to the history objects, possibly limited by privacy restrictions, and only by predefined methods.
- The history objects are immutable from the programmer's perspective and only incremental updates are allowed at the operating system level.
- We support runtime checking of specified safety properties.
- The combination of predefined classes for history objects and contract-dependent adaptations allows flexible adjustments in a safe manner.
- The history object interface specifications guarantee corruption-freedom, since the functional specifications cannot be overwritten or modified.

These advantages are obtained without use of blockchains. But as mentioned, blockchain implementations may be used when underlying middleware or hardware is not reliable, or when the application is financially critical.

The consensus mechanism provided by blockchain is handled differently in our setting. For each contract, the history object is immutable and defines the transactions in an objective manner, outside the contract provider control. Our framework gives language-based support of security, privacy, and safety:

- Security measures such as a blacklist can be added by specifying how to blacklist objects (and similar for whitelisting).
- Privacy can be added by restricting the access to information in the transaction history, defining the *trusted* function for the contract.
- Contract specifications of what is *safe* can be specified for history objects and checked at runtime by these history objects.

This support of security, privacy, and safety is built into the history objects and is therefore useful in a setting where the contract objects and/or its users could be malicious. More advanced security/privacy mechanisms can be introduced by added interfaces and classes.

12. Related Work

In general there are two approaches for verifying smart contracts: dynamic analysis/runtime verification and static analysis/compile-time technique. Runtime verification deals with various techniques for checking whether a running system satisfies or violates certain correctness properties, including those in [16, 20, 25, 42, 46, 19]. Compile-time techniques analyze programs before runtime, either by a fully automatic technique (such as a type and effect system) or by a semi-automatic technique (such as an interactive verification system). In general, compile-time techniques have the advantage of providing guarantees of program properties before runtime. The disadvantage is that fully automatic techniques often involve over-approximation, and formal verification may require non-trivial human assistance. On the other hand, runtime verification has the advantage of more precise analysis, but in general introduces additional runtime overhead. In smart contract platforms like Ethereum, these overheads not only cost time, but also money since they cause additional gas consumption, therefore the cost of smart contract execution increases. As mentioned, our approach with runtime checking of contract invariants by the separate history objects, will not slow down the contract providers (unless they run on the same platform).

A static analysis is said to be syntactic if it is only concerned with the grammatical structure of a program, and semantic if it involves the meaning of grammatically correct programs. Thus semantic analysis is, in general, more powerful in recognizing potential problems in a program. There are many syntactic analysis tools for smart contracts as well as smart contracts written in Solidity (e.g., Solcheck [3], Solhint [4], Solint [5], Solium [6]). Different approaches have been presented that try to identify and fix different types of security vulnerabilities and design patterns [34] in Ethereum smart contracts using semantic analysis techniques. These are summarized below and a comparison to our work is given afterwards.

Luu et al. [33] provided a symbolic execution static analysis tool called OYENTE that analyses Ethereum smart contracts in order to detect bugs. Their tool works directly with EVM bytecode without access to the high-level representation (like Solidity). Also, in [18], a heuristic indicator of control flow immutability has been defined by Fröwis and Böhme for the sake of qualifying loophole risks resulting from modified control flow. They applied this to a number of smart contracts on Ethereum, and found that two out of five smart contracts are not trustworthy.

Mavridou and Laszka [34] introduced a formal semantic model called FSolidM for creating secure smart contracts, that is based on Finite State Machines (FSM). They provided a graphical editor to help simplifying the design of smart contracts in FSolidM. They translate FSMs into Solidity code to support Ethereum smart contracts. They also provided extensions and design patterns to improve the security and functionality of contracts. These extensions and patterns are implemented as plugins that developers can add to a contract automatically.

Many of these semantic approaches are at the bytecode level; therefore they allow verification of compiled contracts. Grishchenko et al. [21], for instance, developed a translation from Solidity to the functional language F*, for which verification tools exist. They have also presented a formalization of a small-step semantics for EVM bytecode in F*. Thus they can compare the F* code resulting directly from Solidity with that resulting via EVM.

Hirai [24] defined a formalization of EVM and used it to prove safety

[3] See https://github.com/federicobond/solcheck
[4] See https://github.com/protofire/solhint
[5] See https://github.com/SilentCicero/solint
[6] See https://github.com/duaraghav8/Ethlint

properties of some smart contracts in the interactive theorem prover Isabelle/HOL. Bhargavan et al. [10] presented a framework to analyze and formally verify Ethereum smart contracts using F*. Their smart contract verification combines two approaches: They start from the translation of Solidity source code into F*, and then using decompilation techniques, they go from EVM bytecode into F* code. An automatic verifier to prove temporal safety properties of Ethereum smart contracts called VerX has been presented by Permenev et al. [41]. VerX is based on reduction of temporal property verification to reachability checking, a symbolic execution engine for a fragment of EVM, and a form of predicate abstraction.

KEVM has been presented by Hildenbrandt et al. [23]. KEVM is an executable formal specification of the EVM's bytecode and stack-based language within $\mathbb{K}$ Framework. The $\mathbb{K}$ Framework is a framework based on rewriting logic to define executable semantic specifications of programming languages. A formal verification tool for the EVM bytecode has been presented in [40]. Park et al. have adopted a complete formal semantics of the EVM called KEVM, and using the K-framework's reachability logic theorem prover to generate a correct-by-construction deductive verifier for the EVM. Furthermore, using EVM-specific abstractions and lemmas they have optimized the verifier and improved its scalability. They have used their verifier to verify various high-profile smart contracts like ERC20 token contracts, Ethereum Casper, and DappHub MakerDAO contracts. In [39] Park, Zhang, and Rosu consider end-to-end verification of the deposit smart contract and find several critical issues as well as also bugs in the Vyper compiler. The work is again based on KEVM and analysis of a state transition system.

Bai et al. [6] propose formal modeling and verification of smart contracts using the SPIN model checking tool. Abdellatif et al. [1] use the BIP (Behavior Interaction Priorities) framework to model smart contracts implementation and verify the correctness. This framework uses timed automata to implement the contract functions. In order to deal with external attacks, they have also modeled the blockchain execution environment. They use the Statistical Model Checking (SMC).

Zakrzewski looks at a formalization of the Solidity language [49]. He points out that the Solidity language has no formal semantics and questions the appropriateness of several of the language constructs, from a verification perspective. The paper gives a proposal for a big-step operational semantics for core elements of Solidity, including modifiers, proposing a clarification/revision of unclear semantical issues.

In [8], Bartoletti, Galletta, and Murgia propose a minimal core calculus, TinySol (for "Tiny Solidity"), for Solidity contracts. The calculus has assignments and a call primitive used to transfer currency and invoke contract procedures. In contrast to our language, the transfer amount is treated as an independent parameter. This construct is inspired by Solidity "external" calls and captures the most paradigmatic aspect of smart contracts like the exchange of digital assets according to programmable rules. They give an operational semantics of the core language and show how to use it to reason about aspects such as reentrancy. Like in our approach, the language is abstract and generalized, but is not considering contract specifications, static verification, security, nor privacy.

In contrast to the works mentioned above, our work does not follow the approach of translating from Solidity or EVM to a language or formalism with verification support, nor are we considering model checking based on an operational semantics or state transition system. Instead, we provide a more abstract programming language for smart contracts, developed to support simple verification at the source level, as encouraged in [3, 49]. The language, its semantics, and verification system are oriented towards simple verification, and we are able to formulate a system for sequential style reasoning in a class-wise manner, together with a composition rule, based on an axiomatic semantics with history-based specifications. Our approach includes a specification formalism for expressing safety properties, and we consider safety rather than temporal properties and do not consider time nor probabilistic methods. Our framework is making use of language-based methods to provide trust, security, privacy, and reasoning control. In particular, flexibility and trust are achieved by predefined protected classes for history objects combined with contract-specific adaptations through a functional specification language.

Compared to earlier work on Creol/ABS [27, 26, 15], our language is novel in the combination of initial guards for methods and first-class futures. This combination gives an expressiveness that allows ordering control with respect to method scheduling, thereby extending the expressiveness of [35]. With respect to reasoning, it gives a simpler event structure than in [15], and as a result, the reasoning system is simpler than the Creol/ABS reasoning systems for programs allowing suspension in the middle of a method.

In [9] Bartoletti et al. propose an approach based on static analysis to improve the performance of blockchains by concurrent execution of transactions, thereby overcoming the limitations of serial execution of transactions

on a blockchain node. The work is rooted in a notion of observational equivalence and swappability of transactions. By means of static analysis of the sets of keys read/written by the transactions, they introduce a static approximation of swappability. They transform a block of transactions into an *occurrence net*, describing the partial order induced by the swappability relation. The resulting parallelization preserves semantical equivalence and leads to more efficient execution. In our setting the history object of a contract may run concurrently on a separate node, not slowing down the contract objects and its users. The efficiency of each contract provider could be improved by means of internal concurrency, following [29].

In a recent paper [2], Ahrendt and Bubel present an approach for verification of smart contracts, considering a subset of Solidity including reentrance. They make use of histories to capture the payment history of a contract. In contrast to our work, this history only covers payment transactions and may only be used for specification and verification purposes. A complication of the reasoning system is the possibility of callbacks, which led to serious verification problems of a majority of the contract examples considered. Verification of these could be made modulo assumptions on absence of callbacks, using special rules for this case. This makes the approach less modular as the presence of callbacks depends on the environment of the considered contract. In our language with guarded methods, callbacks may only happen when the contract invariant is satisfied and therefore do not lead to verification complications. In particular we avoid reasoning about *havoc*) Even when adding the suspending call mechanism [27], which allows callbacks while suspending, reasoning is not more complicated (apart from verifying the invariant at the suspension points), but more events would need to be recorded in the history objects to allow reproducibility. Invariants in [2] may express restrictions on the payment history, but may not express conditions over the communication history. The benefits of our system are related to the fact that our language is oriented towards simple verification and that invariants in our work are expressive enough to avoid pre/postconditions, and the complications related to these in a concurrent setting.

13. Conclusion

We have presented a new approach to lightweight smart contracts where transparency, immutability, and protection against corruption are guaranteed at the source code level, by adding a special kind of protected objects

called history objects that record all the calls and future values generated for each contract. Because of these recorded transactions, a history object can be seen as a ledger, but local to a given contract. The history objects are instantiated from a set of protected classes, and the code of these history classes cannot be redefined by user programmers or contract designers. However, a contract designer can make adaptations of the predefined history classes in a protected manner, through interfaces and functional definitions written in an executable side-effect free specification language. This means that a predefined history class may be specialized with different function definitions through different adaptations, while the methods of the predefined history classes are protected and cannot be changed by programmers, whether on purpose or by accident.

The various history classes provide built-in protection of specified aspects of safety, privacy, security, and transfer of assets. The history objects are separated from the contract provider, and can be used by contract parties through the given interfaces. The approach protects against tampering and fraud at the software level, each history object is immutable and corruption-proof, and a user can observe the contract behavior through the history object to ensure validity. Our approach gives trust and transparency without centralized control when the underlying runtime system can be trusted, without the overhead and cost of blockchain. For increased trust the approach may be combined with blockchain when the runtime system cannot be trusted.

Moreover, we give a theory for formal specification and verification of smart contracts. In particular, our approach supports class-wise verification, which is essential in open distributed systems where the contracts interact in an unknown and open environment. The verification of a class is based on sequential reasoning augmented with effects on the transaction history. These advantages have been achieved by defining a version of a high-level language based on the active object paradigm and interface abstraction. We support reasoning about multiple inheritance and allow subclasses and redefinitions without behavioral restrictions from superclasses. The language has an expressiveness that can be compared to Solidity, but is more high-level, with more abstract communications mechanisms and a more abstract data type language. Method guards can be compared to the require mechanism in Solidity, but give control over the ordering of operations rather than forcing errors. Guards are useful for avoiding undesired contract behavior and avoiding attacks by an adversary. A major difference is that our language has a modular semantics and comes with a specification and verification theory.

A contract class can be verified with respect to an invariant specified in the interface of the history object.

Furthermore, we have shown how formal specifications can be checked automatically by the history objects at runtime, thereby protecting users of a faulty contract provider, as well as protecting the contract provider from illegal users. In addition, we have shown how to deal with security and privacy aspects, something which has been seen as a weakness of traditional smart contracts based on blockchain. We have illustrated the approach on a typical smart contract example, namely an auction system. We have verified the contract implementation and shown various improvements with respect to added safety, security, privacy, and asset protection. In the current state, our framework can be used for executable modeling, prototyping, analysis, verification, and model checking of smart contracts.

The approach may be combined with the notion of dynamic concurrent object groups [29]. This allows a contract service to appear as a single object to the outside while internally consisting of a number of cooperating concurrent objects. From the environment, an object group is treated like a normal object, with the benefit that it can serve many contract users in parallel.

Our framework allows runtime roll-backs, since the transaction history gives sufficient information to rerun a contract service and stop at the last state before the execution of method that results in error. We have shown how protected transfer of assets can be handled directly by the history objects. At compile time, one cannot in general ensure sufficient assets, but one can consider the possibility of errors due to insufficient assets. We have considered reasoning about simple error recovery.

In future work, we may add further mechanisms for error and exception handling and consider efficient implementation of history objects and roll-backs. Furthermore, a way to delete old and irrelevant parts of the transaction history (without violating invariants) would be useful. Finally, the preliminary set of predefined history classes considered here can be extended further. In particular, the security and privacy protection could be improved by more advanced implementations of *SecureHistory* and *PrivateHistory*, and one may investigate the possibility of cryptographic techniques.

Acknowledgment

We thank the reviewers for significant and valuable feedback, which have lead to substantial improvements. Our work was supported by the project IoT-

Sec, Security in IoT for Smart Grids (248113/O70), funded by the Research Council of Norway, under the IKTPLUSS program.

Appendix A. Operational Semantics

We give a brief formalization of the operational semantics of the main elements in our language using a small-step SOS style semantics. A distributed system of active objects can be represented by a multiset of objects and messages, reflecting distribution of components in a network where relative distance and physical placement are abstracted away. A system configuration is then captured by a term like $object_1\ object_2\ object_3\ ...\ msg_1\ msg_2\ msg_3\ ...$ using space as the parallel composition operator, which is associative and commutative. The messages relevant for our setting are invocation events and get events as defined in Section 4.5. We consider here first class futures reflecting future values encapsulated by history objects, as well as local futures reflecting futures values not encapsulated by history objects. Thus there are no separate future objects.

Each rule will involve only one object (and possibly a history object) in addition to consumed and generated messages. This reflects distributed execution. Interleaving semantics is obtained by adding the rule $(A\ B) \longrightarrow (A'\ B)$ if $A \longrightarrow A'$, and similarly $(A\ B) \longrightarrow (A\ B')$ if $B \longrightarrow B'$, where $\longrightarrow$ is the execution transition relation and A and B are subconfigurations. We assume pattern matching modulo associativity and commutativity, which entails that message passing is non-deterministic.

Appendix A.1. Object Representation

An object is represented by a term

$$o : \mathbf{ob}_C(\delta, X)$$

where o is the object identity, C its class, δ its state, and X is the current method execution, represented by either $l :: \overline{s}$ where l is the method call of the executing method and $\overline{s}$ is the remaining list of statements of the executing method, or **idle** when there is no method executing. The state of an object is composed of the attribute state and the local state of the method, and has the form

$$(\rho|\tau)$$

where ρ is the attribute state (including class parameters and system variables such as a runtime stack for local synchronous calls), and τ is the local

state of the method (including the state of explicit and implicit parameters and the state of the local variables). Both ρ and τ are mappings from variable names to values, and we use standard look-up functions $M[v]$ and update functions $M[v \mapsto d]$ for mappings M, where v is a variable name and d is a value. If d is an *error* value, the update operation aborts. We define the state update

$$(\rho|\tau)[v \mapsto d]$$

by $(\rho|(\tau[v \mapsto d]))$ if v is a name in τ, and otherwise by $((\rho[v \mapsto d])|\tau)$ if v is a name in ρ. Similarly, state look-up

$$(\rho|\tau)[v]$$

is defined by $\tau[v]$ if v is a variable name in τ, and otherwise by $\rho[v]$. (Type checking will ensure that each occurrence of a program variable has a binding). We lift this notation to expressions, to denote evaluation of expressions, letting $\delta[e]$ denote the evaluation of an expression e in the object state δ. The evaluation of expressions is standard, inside-out and left-to-right. For instance for a function application $g(e, e')$, we have that $\delta[g(e, e')]$ is $g(\delta[e], \delta[e'])$. The evaluation of an expression with an error handler, say $\delta[e\#e']$, is defined by $\delta[e]$ if the value is not an error, otherwise $\delta[e']$. For convenience, we introduce the notation $\delta[v := e]$ as an abbreviation for $\delta[v \mapsto \delta[e]]$.

As mentioned, our framework gives privacy control of futures in history objects, whereas privacy protection, as well as garbage collection, of separate future objects is challenging [32]. We therefore avoid separate future objects here, letting futures not related to contracts be restricted to local futures. Moreover, we restrict local futures to the write-once/read-once discipline in order to be able to garbage collect futures when read. A future variable may be assigned once (in connection with a call) and then read at most once (by a *get* statement or passed as a parameter in a local call), within the same branch as the call statement. Similarly, a formal future parameter in a local method may be read at most once. Local futures can then be handled by letting the callee send the return value directly to the caller o, by reply messages of the form $get(u, o, d)$ where u is the future identity and d is the returned value. When such a return value is read by a *get* statement the message may be deleted (and thereby avoiding garbage collection).

In order to be globally unique, a future identity can be represented by the identity of the generating object o (the caller) and a locally unique counter

value i, say by the term (o, i). In our setting when future values are contained in history objects, we would also need the identity of the history object z. A future identity in our setting has the form $(o, i)@z$ when it refers to a future value inside a history object z, and (o, i) otherwise. These are referred to as generalized futures and simple futures, respectively. In the rules, j ranges over simple futures and u over both kinds of futures. Only generalized futures represent first-class futures and may be shared between objects, while simple futures may not be returned/passed to other objects and not assigned to other future variables. (This is checked statically.)

More specifically, when a method is called by a contract object o, the future generated is $(o, \mathsf{nextFut})@o.history$ where nextFut is the internal future counter. When o is not a contract object nor a history object, but the *callee* is a contract object, the future generated is $(o, \mathsf{nextFut})@callee.history$, otherwise $(o, \mathsf{nextFut})$. The following function defines the future identity generated in a state δ for a call with o as the caller and o' as the callee:

$$futId(o, o', i) = \textbf{if}\ o \in Contract \wedge o' \notin History\ \textbf{then}\ (o, i)@o.history$$
$$\textbf{else if}\ o' \in Contract\ \textbf{then}\ (o, i)@o'.history\ \ \textbf{else}\ (o, i)$$

The *futId* function identifies which futures are simple and which are generalized ones, and in the latter case, which history object they belong to. Only communication events related to generalized futures will be recorded in the transaction history, since these are the events that involve contracts. Note that *put* and *get* calls will have simple futures. For simplicity, we assume that two contracts are not interacting.

Appendix A.2. Operational Rules

The present operational semantics is similar to the ones presented in [27, 32] apart from the treatment of future variables, returns, and handlers. We assume that programs have been type-checked and are statically correct. The rules for basic statements, shown in Figure A.18, are fairly standard for active object languages. The skip rule shows that *skip* is executed in one step. The assign rule shows how an assignment statement is reflected by the corresponding state update. A handler has no effect if the preceding assignment does not give an error, otherwise it executes the statement list of the handler, without performing the assignment. The if-then and if-false rules show how the execution selects a branch. The end rule says that when there are no remaining statements in a method execution, the object becomes **idle**. An empty statement list is denoted ε, and an empty map is denoted $\emptyset$.

skip:
$$o : \mathbf{ob}_C(\delta, l :: \mathbf{skip}; \overline{s}) \longrightarrow o : \mathbf{ob}_C(\delta, l :: \overline{s})$$

assign :
$$o : \mathbf{ob}_C(\delta, l :: v := e; \overline{s}) \longrightarrow o : \mathbf{ob}_C(\delta[v := e], l :: \overline{s})$$

handle1 :
$$o : \mathbf{ob}_C(\delta, l :: v := e < \overline{s'} >; \overline{s}) \longrightarrow o : \mathbf{ob}_C(\delta[v := e], l :: \overline{s}) \quad \mathbf{if}\ \delta[e] \neq error$$

handle2 :
$$o : \mathbf{ob}_C(\delta, l :: v := e < \overline{s'} >; \overline{s}) \longrightarrow o : \mathbf{ob}_C(\delta, l :: \overline{s'}; \overline{s}) \quad \mathbf{if}\ \delta[e] = error$$

if-true :
$$o : \mathbf{ob}_C(\delta, l :: \mathbf{if}\ b\ \mathbf{then}\ \overline{s1}\ \mathbf{else}\ \overline{s2}\ \mathbf{fi}; \overline{s}) \longrightarrow o : \mathbf{ob}_C(\delta, l :: \overline{s1}; \overline{s}) \quad \mathbf{if}\ \delta[b] = true$$

if-false :
$$o : \mathbf{ob}_C(\delta, l :: \mathbf{if}\ b\ \mathbf{then}\ \overline{s1}\ \mathbf{else}\ \overline{s2}\ \mathbf{fi}; \overline{s}) \longrightarrow o : \mathbf{ob}_C(\delta, l :: \overline{s2}; \overline{s}) \quad \mathbf{if}\ \delta[b] = false$$

end:
$$o : \mathbf{ob}_C((\rho|\tau), l :: \varepsilon) \longrightarrow o : \mathbf{ob}_C((\alpha|\emptyset), \mathbf{idle})$$

simple call :
$$o : \mathbf{ob}_C(\delta, l :: a!m(\overline{e}); \overline{s}) \longrightarrow o : \mathbf{ob}_C(\delta[\mathsf{nextFut} := next(\mathsf{nextFut})], l :: sync(msg); \overline{s})$$
$$msg$$
$$\mathbf{where}\ \ u = futId(o, \delta[a], next(\delta[\mathsf{nextFut}]))$$
$$\mathbf{and}\ \ msg = call(u, o, \delta[a], m(\delta[\overline{e}]))$$

async. call :
$$o : \mathbf{ob}_C(\delta, l :: f := a!m(\overline{e}); \overline{s}) \longrightarrow o : \mathbf{ob}_C(\delta, l :: a!m(\overline{e}); f := futId(o, \delta[a], \delta[\mathsf{nextFut}]); \overline{s})$$

sync. call :
$$o : \mathbf{ob}_C(\delta, l :: v := a.m(\overline{e}); \overline{s}) \longrightarrow o : \mathbf{ob}_C(\delta, l :: a!m(\overline{e}); v := \mathbf{get}\ futId(o, \delta[a], \delta[\mathsf{nextFut}]); \overline{s}) \quad \mathbf{if}\ \ o \neq \delta[a]$$

Figure A.18: Operational rules for basic statements reflecting small-step semantics

start bool :
$$\begin{array}{ll} & call(u, o', o, m(\overline{d})) \\ & o : \mathbf{ob}_C((\alpha|\beta), \mathbf{idle}) \\ \longrightarrow & o : \mathbf{ob}_C((\alpha|\gamma'), call(u, o', o, m(\overline{d})) :: \overline{s}) \ \texttt{if}\ (\alpha, \gamma')[b] = true \\ & \texttt{where}\ m \text{ is bound to } (m, \overline{y}, \gamma, \texttt{when}\ b; \overline{s}) \text{ in } C \\ & \text{and } \gamma' = \gamma[\mathsf{caller} \mapsto o', \overline{y} \mapsto \overline{d}] \end{array}$$

start sfut. :
$$\begin{array}{ll} & call(u, o', o, m(\overline{d})) \quad get(j, o, d) \\ & o : \mathbf{ob}_C((\alpha|\beta), \mathbf{idle}) \\ \longrightarrow & o : \mathbf{ob}_C((\alpha|\gamma'), call(u, o', o, m(\overline{d})) :: \overline{s}) \\ & get(j, o, d) \qquad\qquad\qquad\qquad\qquad\qquad \texttt{if}\ (\alpha, \gamma')[f] = j \\ & \texttt{where}\ m \text{ is bound to } (m, \overline{y}, \gamma, \texttt{when}\ f?; \overline{s}) \text{ in } C \\ & \text{and } \gamma' = \gamma[\mathsf{caller} \mapsto o', \overline{y} \mapsto \overline{d}] \end{array}$$

start gfut. :
$$\begin{array}{ll} & call(u, o', o, m(\overline{d})) \\ & o : \mathbf{ob}_C((\alpha|\beta), \mathbf{idle}) \\ & a : \mathbf{ob}_H(\delta, X) \\ \longrightarrow & o : \mathbf{ob}_C((\alpha|\gamma'), call(u, o', o, m(\overline{d})) :: \overline{s}) \\ & a : \mathbf{ob}_H(\delta, X) \qquad\qquad \texttt{if}\ (\alpha|\gamma')[f] = j@z \ \wedge\ j \in \delta[trans] \\ & \texttt{where}\ m \text{ and } \gamma' \text{ as in Rule start sfut, } H \text{ a history class} \end{array}$$

ret. simp. :
$$\begin{array}{ll} & o : \mathbf{ob}_C(\delta, call(j, o', o, m(\overline{d})) :: \texttt{return}\ e; \overline{s}) \\ \longrightarrow & o : \mathbf{ob}_C(\delta, call(j, o', o, m(\overline{d})) :: \overline{s}) \\ & get(j, o', \delta[e]) \end{array}$$

ret. gen. :
$$\begin{array}{ll} & o : \mathbf{ob}_C(\delta, l :: \texttt{return}\ e; \overline{s}) \\ \longrightarrow & o : \mathbf{ob}_C(\delta, l :: z.put(comp(j, o, o', m(\overline{d}), \delta[e])); \overline{s}) \\ & \texttt{where}\ l = call(j@z, o', o, m(\overline{d})) \end{array}$$

get simp. :
$$\begin{array}{ll} & get(j, o, d) \\ & o : \mathbf{ob}_C(\delta, l :: v := \texttt{get}\ j; \overline{s}) \\ \longrightarrow & o : \mathbf{ob}_C(\delta, l :: v := d; \overline{s}) \end{array}$$

get gen. :
$$\begin{array}{ll} & o : \mathbf{ob}_C(\delta, l :: v := \texttt{get}\ j@z; \overline{s}) \\ \longrightarrow & o : \mathbf{ob}_C(\delta, l :: v := z.get(j); \overline{s}) \end{array}$$

Figure A.19: Operational rules for future-related statements.

A simple call is executed by generating a new future identity (using *futId*) and sending a call message to the callee. The local counter nextFut is updated using a function *next* that gives the next counter value (say by adding one). In case there is a call to be recorded in a history object z, there must be a *put* to z. To handle this, we define a semantic function *sync*

$$sync(call(j@z, o, o', m(\overline{d})) = z.put(call(j, o, o', m(\overline{d}))$$
$$sync(call(j, o, o', m(\overline{d})) = skip$$

Any call to *put* is made by a synchronous call to ensure proper ordering in the transaction history. An asynchronous call is like a simple call except that is assigns the generated future value to the future variable. A non-local synchronous call is like a simple call followed by a **get** statement on the generated future. The rules for method calls are fairly typical for active object systems with two-way message passing, as the history objects are not involved. Local synchronous calls may be handled by adding a run-time stack, or using the technique in [27]. We do not include rules for local synchronous (stack-based) calls nor object generation, since these are not central to the contribution here. Multicast and broadcast can be handled with the technique in [28].

The rules for future-related statements are shown in Figure A.19, and here awareness of the history objects matters. The start rules describe how a new method execution is started, which may only happen when there is a call message to an object o, o is **idle**, and when the guard (if any) is satisfied. The statement list of the called method is prefixed by the call message since this information is needed in order to deal with the history transaction upon return. (This call message label is written in blue for better readability.) A boolean guard is evaluated in the state of the attributes of o combined with the local state of the called method, including the actual parameters. The rule for a method without a guard is similar, just without the if-condition, and is omitted here. A guard on a simple future variable, is satisfied if the future is resolved, which means that there must be a completion message with the future available. The completion message stays since the future value is not yet read. A guard on a generalized is satisfied if the future identity is found in a transaction recorded in the corresponding history object. Thus this rule makes a read access to the history object, but otherwise o behaves as in the case of a simple future (Rule start sfut).

The ret. simp. rule defining method return from a method execution with

a simple future identity, reflects the two-way call/reply paradigm since a simple future cannot be shared. The return from a method execution with a generalized future is stored in the corresponding history object. This is done through a synchronous *put* call to ensure that the appropriate transaction order in the history object. This rule is non-standard since there is an effect on the history object (i.e., a synchronous call on *put* must be handled). An asynchronous *put* call could lead to a different ordering of completions in the history object than in *o*.

A *get* statement on a simple future can be performed when the completion message for this future is available. The future value is assigned to the left-hand-side variable of the *get* statement. For a *get* statement on a generalized future, the future value is obtained from the history object by means of a synchronous *get* call. Note that the synchronous calls in the *ret. gen.* and *get gen.* rules ensure that the order of *comp* events in the history variable will be the same as the order of *comp* events generated by the corresponding contract objects. In order to save time, the *put* may return immediately after being started and continue with the remaining statements since the return value is known (*void*), as exploited in class *SAFEHISTORY* of Figure 9.

Since each update on an object state is done by the state update operation, we note that an object state cannot contain *error* since that will abort the current method execution. In this case a rollback could be made. A formalization of this would be beyond the scope of this paper. For the sake of accountability, we have added *get* events in the transaction history to record which party has looked at which future values. This is done by including a *put* call in the implementation of the *get* method.

Appendix B. Notational conventions for lower case characters

a	object expression (for callee) and also auction contract
b	boolean condition
c	contract
d	data
e	expression
f	future variable
g	function
h	history
i	index
j	simple future identity
k	index
l	call event label
m	method
n	method
o	object
p	party
q	sequence/list
r	result
s	statement
t	transaction
u	future identity
v	variable
w	field
x	variable
y	local variable or formal parameter
z	history object

References

[1] T. Abdellatif and K.-L. Brousmiche. Formal verification of smart contracts based on users and blockchain behaviors models. In 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS), pages 1–5. IEEE, 2018.

[2] W. Ahrendt and R. Bubel. Functional verification of smart contracts via strong data integrity. In S. B. Margaria T., editor, Leveraging Applications of Formal Methods, Verification and Validation: Applications. ISoLA 2020,

volume 12478 of Lecture Notes in Computer Science, pages 9–24. Springer, 2020.

[3] W. Ahrendt, G. J. Pace, and G. Schneider. Smart contracts: A killer application for deductive source code verification. In P. Müller and I. Schaefer, editors, Principled Software Development, pages 1–18. Springer, 2018.

[4] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on Ethereum smart contracts (SoK). In M. Maffei and M. Ryan, editors, Principles of Security and Trust, volume 10204 of Lecture Notes in Computer Science, pages 164–186. Springer, 2017.

[5] F. Baader and T. Nipkow. Term Rewriting and All That. Cambridge University Press, 1998.

[6] X. Bai, Z. Cheng, Z. Duan, and K. Hu. Formal modeling and verification of smart contracts. In Proceedings of the 2018 7th International Conference on Software and Computer Applications, pages 322–326. ACM, 2018.

[7] H. C. Baker and C. Hewitt. The incremental garbage collection of processes. SIGPLAN Not., 12(8):55–59, 1977.

[8] M. Bartoletti, L. Galletta, and M. Murgia. A minimal core calculus for Solidity contracts. In C. Pérez-Solà, G. Navarro-Arribas, A. Biryukov, and J. Garcia-Alfaro, editors, Data Privacy Management, Cryptocurrencies and Blockchain Technology, volume 11737 of Lecture Notes in Computer Science, pages 233–243. Springer, 2019.

[9] M. Bartoletti, L. Galletta, and M. Murgia. A true concurrent model of smart contracts executions. In S. Bliudze and L. Bocchi, editors, Coordination Models and Languages, volume 12134 of Lecture Notes in Computer Science, pages 243–260. Springer, 2020.

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

[11] F. D. Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, and A. M. Yang. A survey of active object languages. ACM Comput. Surv., 50(5):1–39, 2017.

[12] V. Buterin. Critical update re: DAO vulnerability. Ethereum Blog, June, 2016.

[13] V. Buterin et al. Ethereum white paper. GitHub repository, pages 22–23, 2013.

[14] O.-J. Dahl and O. Owe. Formal methods and the RM-ODP. In NWPT'98: Nordic Workshop on Programming Theory, Turku, Finland 1998, 1998. Available as Research Report 261, Dept. of Informatics, Univ. of Oslo (18 pages).

[15] C. Din and O. Owe. Compositional reasoning about active objects with shared futures. Formal Aspects of Computing, 27:551–572, 2014.

[16] J. Ellul and G. J. Pace. Runtime verification of Ethereum smart contracts. In 2018 14th European Dependable Computing Conference (EDCC), pages 158–163. IEEE, 2018.

[17] E. Fazeldehkordi, O. Owe, and T. Ramezanifarkhani. A language-based approach to prevent DDoS attacks in distributed financial agent systems. In A. P. Fournaris, M. Athanatos, K. Lampropoulos, S. Ioannidis, G. Hatzivasilis, E. Damiani, H. Abie, S. Ranise, L. Verderame, A. Siena, and J. García-Alfaro, editors, Computer Security - ESORICS 2019 International Workshops, IOSec, MSTEC, and FINSEC, Revised Selected Papers, volume 11981 of Lecture Notes in Computer Science, pages 258–277. Springer, 2019.

[18] M. Fröwis and R. Böhme. In code we trust? In J. Garcia-Alfaro, G. Navarro-Arribas, H. Hartenstein, and J. Herrera-Joancomartí, editors, Data Privacy Management, Cryptocurrencies and Blockchain Technology, volume 10436 of Lecture Notes in Computer Science, pages 357–372. Springer, 2017.

[19] L. García-Bañuelos, A. Ponomarev, M. Dumas, and I. Weber. Optimized execution of business processes on blockchain. In J. Carmona, G. Engels, and A. Kumar, editors, Business Process Management, pages 130–146. Springer, 2017.

[20] G. Governatori, F. Idelberger, Z. Milosevic, R. Riveret, G. Sartor, and X. Xu. On legal contracts, imperative and declarative smart contracts, and blockchain systems. Artificial Intelligence and Law, 26(4):377–409, 2018.

[21] I. Grishchenko, M. Maffei, and C. Schneidewind. A semantic framework for the security analysis of Ethereum smart contracts. In L. Bauer and R. Küsters, editors, International Conference on Principles of Security and Trust, volume 10804 of Lecture Notes in Computer Science, pages 243–269. Springer, 2018.

[22] Á. Hajdu and D. Jovanović. SMT-friendly formalization of the Solidity memory model. In P. Müller, editor, Programming Languages and Systems, volume 12075 of Lecture Notes in Computer Science, pages 224–250. Springer, 2020.

[23] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, et al. KEVM: A complete formal semantics of the Ethereum virtual machine. In 2018 IEEE 31st Computer Security Foundations Symposium (CSF), pages 204–217. IEEE, 2018.

[24] Y. Hirai. Defining the Ethereum virtual machine for interactive theorem provers. In A. Kiayias, editor, International Conference on Financial Cryptography and Data Security, volume 10204 of Lecture Notes in Computer Science, pages 520–535. Springer, 2017.

[25] F. Idelberger, G. Governatori, R. Riveret, and G. Sartor. Evaluation of logic-based smart contracts for blockchain systems. In J. J. Alferes, L. Bertossi, G. Governatori, P. Fodor, and D. Roman, editors, Rule Technologies. Research, Tools, and Applications, volume 9718 of Lecture Notes in Computer Science, pages 167–183. Springer, 2016.

[26] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, International Symposium on Formal Methods for Components and Objects, volume 6957 of Lecture Notes in Computer Science, pages 142–164. Springer, 2010.

[27] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. Software & Systems Modeling, 6(1):39–58, 2007.

[28] E. B. Johnsen, O. Owe, J. Bjørk, and M. Kyas. An object-oriented component model for heterogeneous nets. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, volume 5382 of Lecture Notes in Computer Science, pages 257–279. Springer, 2007.

[29] E. B. Johnsen, O. Owe, D. Clarke, and J. Bjørk. A formal model of service-oriented dynamic object groups. Sci. Comput. Program., 115-116:3–22, 2016.

[30] E. B. Johnsen, O. Owe, and I. Simplot-Ryl. A dynamic class construct for asynchronous concurrent objects. In M. Steffen and G. Zavattaro, editors, Formal Methods for Open Object-Based Distributed Systems, 7th IFIP WG 6.1 International Conference, FMOODS 2005, volume 3535 of Lecture Notes in Computer Science, pages 15–30. Springer, 2005.

[31] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. Theoretical Computer Science, 365(1-2):23–66, 2006.

[32] F. Karami, O. Owe, and T. Ramezanifarkhani. An evaluation of interaction paradigms for active objects. J. Log. Algebr. Meth. Program., 103:154–183, 2019.

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

[34] A. Mavridou and A. Laszka. Designing secure Ethereum smart contracts: A finite state machine based approach. In S. Meiklejohn and K. Sako, editors, International Conference on Financial Cryptography and Data Security, volume 10957 of Lecture Notes in Computer Science, pages 523–540. Springer, 2018.

[35] O. Owe. Verifiable programming of object-oriented and distributed systems. In L. Petre and E. Sekerinski, editors, From Action Systems to Distributed Systems - The Refinement Approach, pages 61–79. Chapman and Hall/CRC, 2016.

[36] O. Owe, E. Fazeldehkordi, and J.-C. Lin. A framework for flexible program evolution and verification of distributed systems. In S. Hammoudi, L. F. Pires, and B. Selić, editors, Model-Driven Engineering and Software Development, volume 1161 of Communications in Computer and Information Science, pages 320–349. Springer, 2020.

[37] O. Owe and G. Schneider. Wrap your objects safely. Electron. Notes Theor. Comput. Sci., 253(1):127–143, 2009.

[38] Parity Technologies. A postmortem on the parity multi-sig library self-destruct, 2018. `https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/`.

[39] D. Park, Y. Zhang, and G. Rosu. End-to-end formal verification of Ethereum 2.0 deposit smart contract. In S. K. Lahiri and C. Wang, editors, Computer Aided Verification, volume 12224 of Lecture Notes in Computer Science, pages 151–164. Springer, 2020.

[40] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu. A formal verification tool for Ethereum VM bytecode. In Proceedings of the 2018 26th ACM Joint

Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, pages 912–915. ACM, 2018.

[41] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. Verx: Safety verification of smart contracts. In 2020 IEEE Symposium on Security and Privacy, SP, pages 18–20. IEEE, 2020.

[42] C. Prybila, S. Schulte, C. Hochreiner, and I. Weber. Runtime verification for business processes utilizing the Bitcoin blockchain. Future Generation Computer Systems, 107:816–831, 2020.

[43] I. Sergey and A. Hobor. A concurrent perspective on smart contracts. In A. Kiayias, editor, Financial Cryptography and Data Security, volume 10322 of Lecture Notes in Computer Science, pages 478–493, 2017.

[44] soliditylang.org. Solidity documentation, 2019. `https://solidity.readthedocs.io/`.

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

[46] I. Weber, X. Xu, R. Riveret, G. Governatori, A. Ponomarev, and J. Mendling. Untrusted business process monitoring and execution using blockchain. In M. La Rosa, P. Loos, and O. Pastor, editors, Business Process Management, volume 9850 of Lecture Notes in Computer Science, pages 329–347. Springer, 2016.

[47] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper, pages 1–32, 2014.

[48] A. Yonezawa. ABCL: An Object-Oriented Concurrent System. MIT Press, 1990.

[49] J. Zakrzewski. Towards verification of Ethereum smart contracts: A formalization of core of Solidity. In R. Piskac and P. Rümmer, editors, Verified Software: Theories, Tools, and Experiments, volume 11294 of Lecture Notes in Computer Science, pages 229–247. Springer, 2018.