Smart Contracts: Application Scenarios for Deductive Program Verification

Bernhard Beckert, Jonas Schiffl, and Mattias Ulbrich

Karlsruhe Institute of Technology, Karlsruhe, Germany
bernhard.beckert@kit.edu, jonas.schiffl@kit.edu, mattias.ulbrich@kit.edu

Abstract. Smart contracts are programs that run on a distributed ledger platform. They usually manage resources representing valuable assets. Moreover, their source code is visible to potential attackers, they are distributed, and bugs are hard to fix. Thus, they are susceptible to attacks exploiting programming errors. Their vulnerability makes a rigorous formal analysis of the functional correctness of smart contracts highly desirable.
In this short paper, we show that the architecture of smart contract platforms offers a computation model for smart contracts that yields itself naturally to deductive program verification. We discuss different classes of correctness properties of distributed ledger applications, and show that design-by-contract verification tools are suitable to prove these properties. We present experiments where we apply the KeY verification tool to smart contracts in the Hyperledger Fabric framework which are implemented in Java and specified using the Java Modeling Language.

1 Introduction

Smart contracts are programs that work in conjunction with a distributed ledger. They automatically manage resources on that ledger. Multiple distributed ledger platforms supporting smart contracts have been developed, most prominently the public Ethereum blockchain. Smart contracts manage resources representing virtual or real-world assets. Their source code is visible to potential attackers. Therefore, they are susceptible to attacks exploiting errors in the program source code. Furthermore, smart contracts cannot be easily changed after deployment. They need to be correct upon deployment, and formal methods should be used for ensuring their correctness [**?**].

In this paper, we describe the computational model of smart contracts, which makes them an ideal target for deductive program verification. We discuss different notions of smart contract correctness, and the implications for formal verification.

We focus on the Hyperledger Fabric [**?**] architecture. Fabric is a framework for the operation of private, permission-based distributed ledger networks. Smart contracts in Fabric can currently be written in Go, Java, and Javascript. While our concrete verification efforts target Fabric smart contracts written in Java, much of the concepts can be generalized to other programming languages, and also to other smart contract platforms.

The KeY system [**?**], which we used for experiments, is a deductive program verification tool for verifying Java programs w.r.t. a formal specification. KeY follows the principle of design-by-contract, i.e., system properties are broken down into method specifications called *contracts* that must be individually proven correct. Specifications for KeY are written in the Java Modeling Language [**?**], the

de-facto standard language for formal specification of Java programs. For verification, KeY uses a deductive component operating on a sequent calculus for JavaDL, a program logic for Java.

In **??**, we describe an abstract computational model for applications in a distributed ledger architecture. In **??**, we discuss different notions and classes of smart contract correctness w.r.t. that model. Then, in **??**, we describe how properties from these classes can be verified in the KeY tool. Finally, we draw some conclusions and discuss future work in **??**.

2 Distributed Ledger Infrastructure and the Computational Model

Smart contract platforms are complex systems. Their functionality is spread across several layers and components. Some components are by necessity part of every smart contract platform, other components are unique to certain types of smart contract systems.

The correct behavior of a smart contract depends on all components of the distributed ledger architecture. This includes: the implementation of the blockchain data structure, which ensures that the shared history cannot be changed; the consensus and ordering algorithms for creating a single view of the system state; the cryptography modules for chain integrity and the public key infrastructure; and the network layer, which ensures correct distribution of transaction requests and new blocks.

If all these components work correctly, they provide an abstract computational model for the execution of smart contract applications in a distributed ledger system. This computational model can be described as follows: A distributed ledger platform behaves like a (non-distributed) single-core machine which takes requests (in the form of function calls) from clients. The execution of a request (a transaction) is atomic and sequential. The machine's storage is a key-value database in which serialized objects are stored at unique addresses. The storage can only be modified through client requests. The overall state of the storage is determined entirely by the order in which requests are taken. No assumptions can be made about the relationship between the order of requests made by the clients and the actual order of execution. However, it can be assumed that every request is eventually executed. All requests are recorded, even if they do not modify the state or are malformed.

3 Correctness of Smart Contracts

In the previous section, we have described the abstraction provided by smart contract architectures: It behaves like a single-core machine operating on a database storage and taking requests from clients. In this section, we discuss how this abstraction is useful for applying program verification techniques and tools. We give an overview of different classes of smart contract correctness properties and characterize the requirements and challenges for formal analysis that each class entails. The properties are roughly ordered by the effort required to prove them. Existing approaches to verification of smart contracts are given as examples for each class.

3.1 Generic Properties

Generic properties are independent of the concrete smart contract application and its functionality, i.e., there is not need to write property specifications for individual contracts. Typical examples of generic properties are termination for all inputs, absence of exceptions (such as null-pointer dereference), and absence of type errors.

Program properties such as termination are undecidable in general, and proofs may be non-trivial and require heavy-weight verification tools. Nevertheless, many generic properties can be validated by syntactical methods like type checking or simple static analysis. They are less precise than program verification and produce false alarms in case of doubt, but are still very useful in practice. Especially in the context of Ethereum, there is a wide variety of static analysis tools, e.g. [**?**, **?**], that can show the absence of known anti-patterns or vulnerabilities, like the notorious reentrance vulnerability, or inaccessible funds. For Hyperledger Fabric, there exists a tool which statically checks a smart contract for anti-patterns like non-determinism or local state.[1]

3.2 Specific Correctness Properties of Single Transactions

Correctness of a smart contract applications cannot be captured by generic properties alone: There has to be some formal specification which expresses the expected resp. required behavior of a program. Smart contract functions, which are atomic and deterministic in our computational model, are the basic modules of smart contracts (much like methods are basic components of programs), and therefore also the basic targets for correctness verification. The specification of a function consists of a precondition, which states what conditions the caller of the function has to satisfy, and a postcondition expressing what conditions are guaranteed to hold after the transaction (i.e., the function execution).

In case of smart contracts, the precondition should generally be empty because no assumptions about the state of the ledger should be necessary for correctness; furthermore, the values of the call parameters could be chosen by a malicious agent, and correctness properties need to hold for all possible input values.

Examples of specific properties of a single transaction include functional correctness statements (e.g., "the specified amount is deducted from the account if sufficient funds are available, otherwise the account remains unchanged") and statements about which locations on the ledger a transaction is allowed to modify.

An approach to verification of single transaction correctness using the Why3 verification platform has been proposed [**?**]; our own approach using the KeY tool is discussed in **??**.

3.3 Correctness of Distributed Ledger Applications

While transactions are equivalent to individual program functions, a *distributed ledger application* (DLA) is equivalent to a reactive program whose functions can be called by external agents. Informally, a DLA is a part of a smart contract network concerned with one specific task, like running an auction or providing a bank service. More precisely, a DLA is the set of all transactions that can

[1] https://chaincode.chainsecurity.com/

affect a given set of storage locations (including transactions that cannot access a storage location but are used in the calculation of the values being written).

While correctness of the component transactions is a necessary pre-requisite for the correctness of the DLA, there are properties which inherently are properties of transaction traces. They cannot be readily expressed as correctness properties of single transactions. To break them down into a set of single-transaction properties is a non-trivial process. Examples for this class of properties include invariants (e.g., "the overall amount of funds stays the same" for a banking application) and liveness properties giving the guarantee that some condition will eventually be fulfilled. Complex properties of this kind typically are expressed in temporal logic.

4 Verification of Smart Contracts Using the KeY Tool

In this section, we discuss verification of smart contract correctness using the KeY tool. The abstract computational model devised in **??** is an excellent fit for KeY because, in this setting, a distributed ledger application can be viewed as the equivalent of a Java program where single transactions correspond to Java methods. Thus, the KeY tool, which is designed for verifying Java programs, can be utilized for DLA verification, requiring only minor adaptations. These adaptations mostly concern the nature of the storage, since KeY operates on a heap with object references, while the distributed ledger application's storage is a database of serialized objects. Furthermore, due to the unknown order of execution and the fact that different agents operate within the shared program, there cannot be any assumptions as to the contents of the storage or order of transaction execution.

As an example, we consider a Hyperledger Fabric smart contract that implements functionality for simple auctions. It consists of three public methods, which allow clients to (1) start auctions for items they want to auction off, (2) bid for existing auctions, and (3) close an auction (an action that requires administrative credentials):

```
void createAuction(List<Item> items, int minBid, int endTime) { ... }
void bid(int auctionID, int bid) { ... }
void closeAuction(int AuctionID) { ... }
```

4.1 Generic Properties

The KeY tool can be used to verify any generic property. As a heavy-weight verification tool, it is particularly useful for properties that cannot be handled by light-weight tools resp. that require KeY's higher precision to avoid too many false alarms. Examples are program termination and the absence of exceptional behavior (the Java Modeling Language keyword `normal_behavior` can be used to specify that a method terminates without exception).

While constructing proofs for such properties is a non-trivial task in general, typical smart contracts are compact and lack complex control flows. In such settings, proofs of termination and absence of exceptions can be expected to be found automatically by KeY, requiring none or minimal auxiliary specifications.

4.2 Chaincode Transaction Correctness

Verification of Hyperledger Fabric chaincode functions, if written in Java, is possible in KeY. The difference between a normal Java program and our com-

putational model is in the storage: While Java programs operate on a heap, a Fabric Smart Contract operates on an (abstracted) key-value database storing serialized objects. In our experiments, this difference was addressed by an extension of the KeY tool, including an axiomatisation of the read/write interface of the Fabric ledger, a model of the ledger on a logical level, and the introduction of abstract data types for each type of object that is managed by the smart contract [**?**].

In the auction example, one might want to specify the `closeAuction()` method as follows:

```
/*@ ensures read(ID) != null ==> read(ID).closed;
  @ ensures (\forall Item i \in read(ID).items;
  @               i.owner_id == read(ID).highestBidderID);
  @ modifies read(ID);
  @*/
void closeAuction(int ID) { ... }
```

This JML specification is somewhat simplified for readability; the `read` function is an abstraction for accessing the ledger, i.e., reading and deserializing the object at the given location. The specification states that, if the auction object at `ID` is not null, then after execution of the `closeAuction()` method the `closed` flag must be correctly set; furthermore all items in the auction must belong to the highest bidder (as indicated by the `owner_id` attribute). The `modifies` clause states that only the object at the location specified by `ID` may be changed by the transaction, ensuring that no unexpected side effects are possible. This method contract can be loaded and proven using our smart-contract version of KeY; the logical rules necessary for handling the data types stored on the ledger (in this case, auctions, items, and participants) are created automatically. The proof requires some user interaction, since the new rules have not yet been included in the automation mechanism of the prover.

There exists a comparable approach for using KeY to verify Ethereum smart contracts [**?**].

4.3 Correctness of Distributed Ledger Applications

More complex properties can be reasoned about in KeY using class invariants, two-state invariants, and counters, thereby reducing complex properties of transaction traces (including temporal logic properties) to KeY's method-modular approach. For example, the specification of the auction application could state that, as long an the auction is open, the items that are offered still belong to the auctioneer:

```
//@ invariant (\forall Auction a; !a.closed ==>
                    (\forall item i \in a.items;
                          i.owner_id == a.auctioneer_id));
```

If every bidder has to deposit the funds for their bid in the auction, the specification could state that as long as the auction remains open, the sum of the funds in the auction remains the same or increases, but never decreases. This can be expressed with a history constraint:

```
//@ constraint (\forall Auction a; !a.closed ==>
                     \old(a.funds) <= a.funds);
```

Though this constraint can easily be expressed in the Java Modeling Language, proving in KeY that a smart contract conforms to this specification is currently

infeasible due to the large amount of user interactions that is necessary to close the proof, and due to the inefficiencies of our current approach regarding the handling of reading from and writing to the ledger.

5 Conclusion and Future Work

We have outlined the setting in which deductive program verification of distributed ledger applications takes place and shown that the KeY verification tool is suitable to prove different classes of correctness properties which are interesting in smart contract platforms.

The extensions to KeY which enable verification of Hyperledger Fabric smart contracts are still in a prototypical state. Further improvements are necessary to improve scalability and enable proofs of more complex properties.