Verification of Smart Contract Business Logic

Exploiting a Java Source Code Verifier

Wolfgang Ahrendt[1], Richard Bubel[2], Joshua Ellul[3], Gordon J. Pace[3], Raúl Pardo[4], Vincent Rebiscoul[5], and Gerardo Schneider[6]

[1] Chalmers University of Technology, Gothenburg, Sweden, ahrendt@chalmers.se
[2] Technische Universität Darmstadt, Darmstadt, Germany, bubel@cs.tu-darmstadt.de
[3] University of Malta, Msida, Malta, joshua.ellul@um.edu.mt, gordon.pace@um.edu.mt
[4] Inria, Lyon, France, raul.pardo-jimenez@inria.fr
[5] Ècole Normale Supèrieure de Lyon, Lyon, France, vincent.rebiscoul@ens-lyon.fr
[6] University of Gothenburg, Gothenburg, Sweden, gerardo@cse.gu.se

Abstract. Smart contracts have been claimed as a means of building an additional layer of trust between parties by providing a self-executing equivalent of legal contracts. And yet, code does not always perform what it was originally intended to do, which resulted in losses of millions of dollars. Static verification of smart contracts is thus a pressing need. This paper presents an approach to verifying smart contracts written in Solidity by automatically translating Solidity into Java and using, KeY, a deductive Java verification tool. In particular, we solve the problem of rolling back the effects of aborted transactions by exploiting KeY's native support of JavaCard transactions. We apply our approach to a smart contract which automates a casino system, and discuss how the approach addresses a number of known shortcomings of smart contract development in Solidity.

1 Introduction

Blockchain is a *distributed ledger*, running in a decentralised manner on a network of devices that allows for the exchange of data in a trusted manner. Such *values* may be stored and modified without the need for a centralised trusted authority; trust is established through distributed collaboration following specific protocols. *Cryptocurrencies*, particularly *Bitcoin* [15], was the first proposed application of blockchain. A smart contract platform built on top of blockchain, initially proposed by Ethereum[7], enables for blockchain to be used for many other applications besides cryptocurrencies.

Smart contracts are software programs that are openly stored in the blockchain (they can be read and used by anyone), and—as everything else in blockchains—are permanent and cannot be altered. Their execution is performed by "workers" (commonly known as *miners*) that earn some cryptocurrency in return for their work. A smart contract typically offers means of invoking its functionality so end users can transfer data and cryptocurrency to the contract. The contract is effectively the logic to manage these invocations and execute the corresponding instructions that manipulate the local bookkeeping of data (including the cryptocurrency). Underlying a smart contract lies

[7] https://www.ethereum.org

1. *The casino owner may deposit or withdraw money from the casino's bank as long as the bank's balance never falls below zero.*
2. *As long as no game is in progress, the owner of the casino may make available a new game by tossing a coin and hiding its outcome. The owner must also set a participation cost of choice for the game.*
3. *Clauses 1 and 2 are constrained in that as long as a game is in progress, the bank balance may never be less than the sum of the participation cost of the game and its win-out.*
4. *The win-out for a game is set to be 80% of the participating cost.*

Fig. 1. Excerpt from a legal contract regulating a coin-tossing casino.

a description, and prescription, of an agreement between different parties in order to automate the regulated exchange of value and information over the internet.

The promise of smart contract technology is to diminish the costs of contracting, enforcing contractual agreements, and making payments, while at the same time ensuring trust and compliance, all in the absence of a central trusted authority. Such executable legal contractual agreements suffer from some drawbacks: (i) it is not easy to ensure that the smart contract complies with the legal contractual obligations that the program is intended to implement; and (ii) it is not easy to ensure the correctness of smart contracts. In this paper we focus only on the latter aspect. Consider the legal contract shown in Figure 1 regulating how a simple casino should make a coin-tossing game available to players. A smart contract implementing this legal contract would carry out concrete actions to ensure that the legal contract is never violated. For instance, clause 3 requires that while a game is in progress, there is always enough money available to pay in case the player wins. This could be achieved by allowing a game to start only if there is enough money to pay for a win, and then to disallow withdrawals which result in not enough money left to pay. Or more radically by preventing the casino from withdrawing any money during a game. Either way, we should be able to *prove* that our implementation satisfies the invariant required by such clauses.

Smart contracts are programs, and as such they are vulnerable to bugs just as any other software. Errors may have many causes, like out of range numbers, unintuitive language feature semantics, or intricate mismatches between internal bookkeeping (in the local data) and external bookkeeping (in the blockchain), to name a few. Erroneous behaviour may be intended, explicitly provoked by malicious contract creators, or exploited by opportunists. Bugs in smart contracts may result in massive losses in an irreversible way (as blockchain transactions are permanent, and no authority has the power to undo them). Recent multi-million Ethereum bugs[8] have shown that this is indeed an issue researchers and practitioners should take seriously [3].

In this paper we focus on the verification of smart contracts written in Solidity[9], by translating them automatically into Java. By targeting Java, our translation can exploit the similarities between the contract-oriented and the object-oriented paradigms, and make use of existing verification tools. We use the deductive source code verifier KeY [2] to verify the translated program since it is among the most powerful verification

[8] https://www.theguardian.com/technology/2017/nov/08/cryptocurrency-300m-dollars-stolen-bug-ether
[9] https://solidity.readthedocs.io

tools for object-oriented languages, and specifically, it has native verification support for translations and their abortion, allowing to model the rolling back of program effects. We apply our approach to a case study consisting of a Casino smart contract.

The paper is organised as follows. Section 2 gives some background on smart contracts and the deductive verification tool KeY. In Section 3 we present our Solidity to Java translation. Section 4 is concerned with the verification of the translated Java programs using KeY. Section 5 introduces our case study. We discuss scope and limitations of our approach in Section 6, followed by related work and a conclusion.

2 Preliminaries

2.1 Smart Contracts in Solidity

Since smart contracts are deployed on a blockchain (or some other form of distributed ledger technology) which typically enforces immutability of deployed smart contract code and also due to the critical nature of applications they are often employed for, a different mind set to traditional programming is required [6]. Ethereum's virtual machine provides a 'one world computer' abstraction, the Ethereum Virtual Machine (EVM) [16], an abstract machine that executes transactions atomically whereby a transaction is an action initiated by a smart contract user. The predominant language used to write Ethereum smart contract code is Solidity.

A deployed Ethereum smart contract has an associated unique address, can own Ether (Ethereum's cryptocurrency), and transfer Ether to other addresses which may be other contracts or user accounts. Being Turing complete, the EVM needs to cater for code which may not terminate or takes an unacceptably long time to execute. To get around this, the EVM implements a notion of *gas* — a cost (in Ether) for the execution of each EVM bytecode instruction. If the amount of gas associated with a particular transaction is not fully paid for, then execution of the smart contract stops and the altered state within the transaction is reverted to the original state as it was upon initiation. This ensures that all transactions terminate, and that computationally expensive functionality is financially prohibitive, avoiding attempts to overload the Ethereum execution engine.

Listing 1.1 shows an excerpt of our own Solidity contract[10] modelling the casino contract from Figure 1. In particular, the implementation of function `removeFromPot` which allows the casino to withdraw money from the casino's bank when invoked is presented. The logic within the function is simple — it reduces the internal state variable `pot` which keeps track of how much money lies in the casino's bank (using an unsigned 256 bit integer) and transfers the requested amount using the **`transfer`** method to the caller of the function — **`msg`** is a variable representing the message invoking the transaction and **`msg.sender`** is the transaction initiator's address. It is worth noting that on Ethereum, function calls are atomic (though still reentrant), in that they execute to completion (whether successful or not) before another function call can be invoked.

To ensure that the function can only be invoked by the casino owner and not during an active game, the code uses two modifiers `byOperator` and `noActiveBet`, which add in-line checks. The definition of the `byOperator` modifier is also shown in Listing 1.1.

[10] See https://git.io/fx6cn.

```
contract Casino {
  private uint256 pot = 0;
  private address operator;
  ...
  function removeFromPot(uint256 value) public byOperator noActiveBet {
    pot = pot - value;
    msg.sender.transfer(value);
  }

  modifier byOperator() {
    require (msg.sender == operator);
    _;
  }
  ...
}
```

Listing 1.1. Solidity code to withdraw money from the casino pot and definition of a modifier to ensure that a function can only be invoked by the owner of the casino.

It modifies any function it is applied to (here it has been applied to `removeFromPot`) such that it executes the original function code where the placeholder `_;` is specified. The modifier `byOperator` will thus ensure that the transaction initiator is indeed the casino operator, using a **`require`** statement (one type of exception raising convenience function provided which checks if a condition holds, or otherwise raises an exception), and then executes the original function code. Internally, the **`require`** statement triggers the Solidity command **`revert`** which raises an exception if the condition does not hold.

It is worth noting that if the **`transfer`** function fails (for example due to insufficient available funds being available in the contract) then it will raise an exception and abort the transaction reverting the state (including variable values) back to their original values as at the beginning of the invocation. Solidity also provides a **`send`** function which, upon failure, will not raise an exception but returns a boolean success response.

Functions are tagged by annotations indicating their visibility in Solidity — defining from where calls can be made: **`private`** only from functions within the contract; **`internal`** from functions within the contract or from deriving contracts; **`external`** only from external contracts (or using a contract interface transaction rather than a function call); or **`public`** from anywhere.

2.2 Deductive verification with KeY

We use the KeY system [2] to verify the Java programs (obtained from the original Solidity contracts) to be correct with respect to their specification. The Java Modeling Language (JML) [12], is used to write class invariants and method specifications.

JML specifications are embedded into Java source code as Java comments. Any comment starting with `//@` or `/*@` marks the start of a JML specification. Consequently, standard Java tools like compilers, simply ignore JML specifications, while JML aware tools can distinguish Java comments from JML specifications and make use of them.

Listing 1.2 shows a class `Account`, which implements a bank account. It consists of two integer fields `accountNr` and `balance` as well as the method `transfer`, which takes as arguments the target account (parameter `to`) and the `amount` to be transferred.

```
1  public class Account {
2     /*@ public invariant accountNr >= 0 &&
3            (\forall Account a; a != this; a.accountNr != this.accountNr); *@/
4     private /*@ spec_public @*/ int accountNr;
5
6     //@ public invariant balance >= 0;
7     private /*@ spec_public @*/ int balance;
8
9     /*@ public normal_behaviour
10      @ requires amount >= 0 && to != this;
11      @ requires this.balance >= amount;
12      @ assignable this.balance, to.balance;
13      @ ensures this.balance == \old(this.balance) - amount;
14      @ ensures to.balance == \old(to.balance) + amount;
15      @ ensures \result == true;
16      @*/
17    public boolean transfer(Account to, int amount) {...}
18 }
```

Listing 1.2. Java source code annotated with JML specifications.

The class is annotated with two *JML invariants*. JML invariants specify properties of objects that have to be established by the constructor and to be preserved by all methods. They are marked by the keyword **`invariant`** and followed by the actual property written as boolean typed *JML expression*. JML expressions are a superset of side-effect free Java expressions with additional operators like quantifiers **`\exists`** and **`\forall`**. The first invariant (lines 2–3) states that account numbers are unique, while the second (line 6) restricts the value of field `balance` to be non-negative.

Lines 9–16 contain `transfer`'s *JML method specification*. The method's preconditions are marked by **`requires`**, which is followed by a boolean JML expression. If the caller ensures that the preconditions evaluate to true at invocation time, then the method guarantees that (i) it terminates normally, i.e., without throwing an exception (line 9), (ii) in its final state the postcondition (keyword **`ensures`**) holds (line 13–15) and (iii) that at most the values of the fields listed in the **`assignable`** clause (line 12) have been changed. Multiple **`requires`** and **`ensures`** clauses are conjunctively combined, and many method specifications can be connected using **`also`**. Complementary to **`normal_behavior`** there are **`exceptional_behavior`** specifications stating which exceptions are thrown under which conditions as well as assertions about the post state.

For convenience, JML defines a few defaults. For instance, by default all fields, parameters and return values of reference type are not **`null`**. Further, there is an implicit pre- and postcondition **`\invariant_for(this)`** for each method specification stating that the method has to preserve the invariant of the **`this`** object.

To verify that a Java program satisfies its JML specification, KeY translates Java and JML into a program logic called *Java Dynamic Logic* [2]. The formula is then proven using a sequent calculus and symbolic execution. Symbolic execution is seamlessly integrated as sequent calculus rules. KeY supports modular reasoning by using a method specification to symbolically execute a method invocation statement, instead of inlining the method's body. A program in KeY is thus proven to be correct by verifying one method at a time. The use of method specifications makes the approach modular.

Finally, in the context of the current work, it is important to note that KeY not only supports full sequential Java, but also JavaCard, a Java derivative which features a

transaction mechanism including rollback of interrupted transactions [2]. The fact that KeY natively supports transaction verification enables us to deal with rollback, which is the mechanism used by the EVM to deal with failure in transactions.

3 Translation to Java

We describe here our translation of Solidity contracts into Java. First, we describe the challenges in realising a semantics preserving translation from Solidity to Java. Then we explain our translation in detail. Some challenges (e.g., challenge 1) are common to all smart contract languages, whereas others (e.g., challenge 4) are Solidity specific.

1. *Distributed ledger.* Solidity contracts execute on the blockchain where all transactions are recorded and the balance of all contracts is maintained. Functions such as `transfer` use the blockchain to record exchanges of money between contracts. Neither the distributed ledger nor the functions operating over it exist in the Java runtime and are thus to be implemented separately.
2. *Message passing.* Solidity contracts may trigger the execution of functions in other contracts through *external calls* using message passing. The message not only triggers the right functionality (by naming the function to be executed), it also carries further information such as the address of the message sender and funds sent with the message. So, simply encoding Solidity function calls as Java method calls does not work as the extra information has to be passed within the method calls.
3. *Revertible transactions*. Handling of messages in smart contracts takes the form of a transaction, and failures throughout its execution result in a rollback, reverting the state to what it was at the beginning of the call. Unless explicitly handled, such failures propagate even when they happen in further function calls within the same contract or external ones. Such failures can occur indirectly due to attempts to transfer unavailable funds, or directly through the `revert` command, possibly encapsulated within other instructions such as `require`. Java has no built-in notion of such revertible transactions, and their interaction with the underlying ledger further complicates their encoding.
4. *Bounded datatypes*. Although the EVM uses a 256-bit stack, Solidity provides a family of bounded datatypes, such as the unsigned 256-bit integers `uint256` and signed 24-bit integers `int24`, none of which have direct equivalents in Java. These datatypes have over- and underflow semantics, e.g. using a `uint256`, subtracting 5 from 4 would result in $2^{256} - 1$. These datatypes are common sources of errors and many smart contract vulnerabilities are due to insufficient checks for exceeding bounds, hence, these are to be carefully modelled in the translation.
5. *Function annotations and modifiers*. Solidity allows functions to be tagged by visibility and other built-in annotations, but also with user-defined modifiers. Visibility annotations define access to contract functions, while built-in annotations include `pure` and `view` (indicating that a function will and may not change the contract's state) and `payable` (indicating that messages invoking the function may include transfer of funds with the smart contract as beneficiary together with the message). Furthermore, as discussed earlier, functions can also be annotated by user-defined modifiers, effectively code transformations, which are normally used to include re-

curring snippets of code into functions. Java only supports visibility modifiers and even these do not have a direct correspondence with their Solidity counterpart — the rest remain to be encoded in the translation.

6. *Fallback function*. The message-passing invocation model used by Solidity allows for the handling of messages invoking functions which are not defined in the contract using a fallback function. A contract tries to match the message function name with the functions defined in the contract to which the message is sent, but if none match, the contract's fallback function is invoked. For instance, if a contract at address `addr` does not define a function `f`, then any call to `addr.f` will result in the invocation of the fallback function at address `addr`. A common instance of this is that unless a smart contract explicitly defines a **`transfer`** function (to receive funds), whenever another contract tries to send it funds through `addr.`**`transfer`**, the fallback is invoked. This means that, in such a case, unless the fallback function is annotated as payable, the contract cannot receive funds. This message handling mechanism is completely absent in Java, and requires to be explicitly modelled at different points of the translation.

We explain now how our automated translation addresses the above challenges in order to preserve the semantics of the original Solidity contract.

(i) **The distributed ledger's functionality is abstracted as a public Java class.** To be able to model the environment of the smart contract — the blockchain system on which it runs — we abstract it as a public class, **`Address`**, providing the functionality of the distributed ledger on which the Solidity contracts operate (challenge 1). Thus, the distributed ledger is modelled as several **`Address`** objects that interact using the same functionality as in Solidity's blockchain. The class manages the `balance` of the corresponding Solidity contract and supports methods to `send` and `transfer` modelling what happens at the back of the scenes when the corresponding Solidity functions are invoked. Through this class, the functionality of the `payable` annotation, is also handled, transferring the requested amount from the caller to the callee.

(ii) **Built-in datatypes become public Java classes.** To address challenge 4, the functionality of the Solidity datatypes not available in Java has been replicated in Java interfaces and classes. We end up having multiple classes implementing an interface to support different ways of data handling e.g. should an over- or underflow trigger an exception (used when we want to verify that no exceptions are raised), or should it replicate the semantics of Solidity bounded integers (used in the rare cases when the smart contract may use over- and underflow in its functionality). For instance, the interface **`Uint256`** comes with the `Uint256int` and `Uint256BigInteger` classes to model Solidity's datatype **`uint256`** (see Section 4 for more details). Apart from providing the Solidity operators on these types (e.g. addition and multiplication for integers), the interface is also used to specify generic JML class invariants and method specifications.

We also provide Java implementations that model information about a transaction, a message and a block which are provided by Solidity as global variables accessible from within any function call. This behaviour is replicated by making the transaction, message and block information available as attributes (respectively `tx`, `msg` and `block`) in every contract class and which are updated upon every external function call.

(iii) **Solidity contracts are modelled as Java classes.** Every Solidity contract is translated into a Java class extending the `Address` class in order to have the Ethereum specific features (address where it resides, its balance), includes method definitions to handle `require` throwing an exception to deal with rollback, and includes the state variables of the Solidity contract as class attributes.

(iv) **Contract functions are modelled as methods in the contract class.** In order to translate Solidity function definitions into Java, we must address: (i) annotations; (ii) modifiers; (iii) transaction information and (iv) exception handling. Listing 1.3 shows the Java template generated from a definition of a function `f` with parameters p_1, p_2, etc., the content of which is explained below. Note that from Solidity function `f`, two Java functions are created: one also called `f`, which performs all required checks and then executes the original body of the function; and another function `call_f`, which is the function to be accessed and which adds the necessary machinery to handle exceptions, transaction information, etc.

Visibility annotations `public`, `private` and `internal` are mapped to Java visibility annotations, but `external` (which allows only external calls to the function) has no corresponding annotation in Java and is omitted. Internal uses of such functions would fail at the compilation stage, thus the translation is no less safe. The annotation `payable` is implemented by using the functionality provided by the Java Ethereum model. The visibility annotation in Listing 1.3 is derived from that used in the Solidity contract.

As for user-defined modifiers, we limit our automated translator to deal with modifiers which just inject code before the function's body. Each such modifier is transformed into a method which just executes the code to be injected, and which is invoked at the beginning of the main function call (see Listing 1.3).

Transaction, message and block information is available in Solidity as global variables whenever a function is called. We address this in the Java translation by encoding them as additional parameters to the `call_f` function. To handle failing transactions, Java exceptions are used (since catching exceptions is not possible in Solidity yet, there is no contract code in `catch`). Upon catching an exception, we use the JavaCard transaction rollback mechanism (supported by KeY) to undo the effects of the transaction so far (see the `JCSystem.*` calls appearing in lines 4, 6 and 9 in Listing 1.3).

(v) **Fallback function**. If a Solidity contract has a fallback function defined, then it is translated as described above. If not, we emulate the Solidity compiler and define an empty payable fallback method.

(vi) **Function calls**. Function calls are handled differently depending on whether they are internal or external, as determined at translation time. External calls performed as `A.f();` (or using the Solidity `call` mechanism) are translated as calls to `A.call_f(..., msg, block, tx)` in the corresponding contract class, defaulting to the fallback function if no such function is defined. In contrast, internal calls are simply translated as direct calls to method `f(...)` in the contract class.

We implemented our translation in the tool JAVADITY[11]: it takes a Solidity contract and gives a Java file that can be enriched with JML specifications to be verified with KeY.

[11] See https://github.com/rebiscov/Javadity.

```
visibility_annotation return_type call_f(p1, p2, ..., Message _msg, Block _block, Transaction _tx) {
    msg = _msg; block = _block; tx = _tx;
    try {
      JCSystem.beginTransaction(); // Only for verification purposes
      return this.f(p1, p2, ...);
      JCSystem.commitTransaction(); // Only for verification purposes
    } catch (Exception e) {
      System.out.println(e);
      JCSystem.abortTransaction(); // Only for verification purposes
    }
}

visibility_annotation return_type f(p1, p2, ...) {
    this.user_defined_modifier1();
    this.user_defined_modifier2();
    ...
    this.payable();
    // Translated Solidity function code
}
```

Listing 1.3. Methods in contract class for each function in Solidity contract.

Example 1. Consider the Solidity function `removeFromPot` (shown in Listing 1.1). We define, for the sake of this example, a specification for the previous function that includes the following three preconditions: (i) only the operator can remove from the pot, (ii) the value to be removed may not exceed the current value of the pot, and (iii) no game may be in progress (the game state must be either idle or available); and three postconditions: (i) the variable `pot` is reduced by the amount withdrawn, (ii) the caller's balance is increased by this amount, and (iii) the contract's balance is reduced by the withdrawn amount. Furthermore, only the variable `pot` and the balances of the caller and the casino smart contract may change as a result of calling this function.

Upon applying the translation defined in this section (which is automatically carried out by JAVADITY), we obtain the Java implementation shown in Listing 1.4. We (manually) enrich the implementation with the JML specification (lines 2–9 in Listing 1.4) corresponding with the requirements above. Lines 2–4 correspond to the preconditions, lines 6–8 correspond to the postconditions and line 5 includes an assignable clause indicating the variables that may be modified during the execution of the function.

4 Verification with KeY

In this section we outline the idea and principal approach for two aspects of the specification and deductive verification of the Java translations of Solidity contracts.

Unsigned integer of 256 bit length. As explained in Section 3, Solidity's scalar `uint256` datatype is mapped to the interface type `Uint256` in Java. The interface provides all the arithmetic operations and comparisons needed which we specified accordingly in JML. To specify the interface in an efficient manner, we used JML's ghost fields, i.e., fields that only exist on the specification level and not on the implementation level. The advantage of ghost fields is that they can be declared for interfaces as instance fields to be implicitly present in any implementing class. Using a ghost field allows us to relate the interface type to an abstraction and to use the abstraction in other specifications.

```
/*@ private behaviour
  @ requires operator.eq(msg.sender);
  @ requires \invariant_for(value) && value.gr(Uint256.ZERO) && value.leq(pot);
  @ requires state == State.IDLE || state == State.GAME_AVAILABLE;
  @ assignable pot, msg.sender.balance, this.balance;
  @ ensures pot.eq(\old(pot.sub(value)));
  @ ensures msg.sender.balance.eq(\old(msg.sender.balance.sum(value)));
  @ ensures this.balance.eq(\old(this.balance.sub(value)));
  @ ...
  @*/
private void removeFromPot(Uint256 value) throws Exception {
  // Modifiers
  this.byOperator();
  this.noActiveBet();
  // Requires
  this.require(value.gr(Uint256.ZERO) && value.leq(pot));
  //Function code
  this.pot = this.pot.sub(value);
  msg.sender.transfer(this, value);
}

/*@ ... @*/
public void call_removeFromPot(Uint256 value, Message _msg, Block _block, Transaction _tx) throws
    Exception {
  msg = _msg; block = _block; tx = _tx;
  try {
    JCSystem.beginTransaction();
    this.removeFromPot(value);
    JCSystem.commitTransaction();
  } catch (Exception e) {
    JCSystem.abortTransaction();
  }
}
```

Listing 1.4. Java translation of the `removeFromPot` function.

We use a ghost field called `_value` of the JML type `\bigint` to model the value as integer. JML's `bigint` datatype represents the mathematical whole numbers. An additional invariant restricts the range of the the ghost field to the range of Solidity's `uint256` datatype. The method specifications can then describe their effect with respect to the ghost field `_value`. Listing 1.5 shows an excerpt of the specification. The expression `\dl_MAXUINT256()` refers to the maximal value of the `uint256` datatype. The specification for the addition (method `sum`) specifies the result in relation to the ghost field `value` and takes care of overflow issues.

To allow the reasoning about `Uint256` to be efficient and to a large degree automatic, the classes using this interface had to be enabled to treat it more similar to a primitive type than a reference type. To achieve this the immutability of the instances of this type needs to be exploited. This is until now not directly supported by KeY and will be added in a future release as additional contribution of our work.

To clarify the issue and solution, assume a class `C` has a field `f` of type `Uint256`. The invariant of class `C` will include the boolean expression `\invariant_for(this.f)` to assert the range restrictions. During verification of each method of class `C` we have in particular to show that its invariant is preserved. This can become tedious as it involves unpacking the invariant of `this.f` even though the method did not reassign any value to `f` and thus because of the immutability could not possibly have changed the validity of `\invariant_for(this.f)`. Exploiting the knowledge about immutability

```
public interface Uint256 {
 /*@ private instance invariant _value >= 0 && _value <= \dl_MAXUINT256();
   @ private final instance ghost \bigint _value; @*/

 /*@ private normal_behavior
   @ requires \invariant_for(value);
   @ ensures \result._value == (this._value+value._value > \dl_MAXUINT256() ?
   @    ((\bigint)-1)*\dl_MAXUINT256() - 1 : (\bigint) 0) + this._value + value._value;
   @ ensures \invariant_for(\result);
   @ accessible _value, value._value;
   @ assignable \strictly_nothing;
   @ ...
   @*/
 Uint256 sum(Uint256 value) throws Exception;
}
```

Listing 1.5. Excerpt from the `Uint256` interface specification.

allows the prover to quickly determine that no operations are able to invalidate the respective invariants. In our proof of concept we simulated this feature by specifying the dependencies of the invariants accordingly via so called **`accessible`** clauses. Due to the not yet implemented support for immutability in KeY, we are currently not able to prove the correctness of our accessible clauses, but can make use of them when proving.

Support for Solidity's state rollback. To provide support for the Solidity's rollback in case of exceptions, the translation makes use of JavaCard's transaction mechanism with explicit commit and abort calls. Note again that KeY supports verification of code using revertible JavaCard transactions [2].

In order to model unexpected failures (by external events and not visible by program semantics), we generalise the JML specifications of the methods such that they allow normal as well as exceptional termination. For these methods a wrapper method `call_m(..)` is created (see Section 3) which wraps the call to `m(..)` using JavaCard transactions. For the JML specification of the wrapper method, we need to distinguish between the commit and abort case. For this we use a boolean ghost field (specification only field) that is true if the abort case has been triggered and false otherwise.

JavaCard's transaction mechanism is API based. `JCSystem.beginTransaction()` starts a transaction and any code until a `JCSystem.commitTransaction()` or `JCSystem.aboutTransaction()` is symbolically executed on a copy of the original heap. In case of a commit the copy replaces the original heap; otherwise, the copy is discarded and the original heap is used instead, thus rolling back the changes in case of an abort.

5 Case Study: Casino Contract

As our case study, we use a Solidity contract modelling a casino (whose legal contract is described in Figure 1). The casino manages a pot represented as a **`uint256`** value representing the amount of ether that can be won in a game. The game is essentially a coin toss. A player places a bet on the outcome, transfers her stake to the contract and records the amount. If the prediction of the player is correct, the pot is transferred to her wallet, otherwise the money the player has bet is added to the pot.

The Solidity contract is translated to a Java program by our tool, and annotated with JML specifications, which describe the full functional behaviour of the contract[12]. In particular, an invariant which states that the balance of the contract is equal to the amount in the pot if no bet is currently placed; otherwise, the contract's balance equals the sum of the ether in the pot and the player's wager.

We verified that a representative selection (meaning that all supported features occur) of methods satisfy their contract and preserve the stated invariant. In particular, we verified for the methods `call_closeCasino` and `call_removeFromPot` that they behave correctly w.r.t. the rollback semantics in case of exceptions. For instance, the proof of `call_removeFromPot` required around 22.000 rule applications of which 207 were interactive. The most critical rule applications were target unpacking of parts of the class invariant. The rationale is to only unpack those parts whose property is required to prove a property, e.g., if the fields occurring in a conjunct of an invariant have been changed by the method and the conjunct has to be shown to be reestablished. For the unchanged parts, dependency contracts are used which exploit the fact that if a formula does not depend on changed parts of the heap then it cannot be invalidated.

The verification effort is rather straightforward. It requires some tedious but trivial interactions due to the `Uint256` datatype being modelled as interface. The necessary rule interactions are less than 1.5% of all rule applications (the peak proof requires 1.4% interactive steps). Thereof the vast majority of interactive rule applications is unpacking of class invariants. This could be easily automated by enforcing the unpacking of invariants before method contracts are used. Some very few require to prove or make use of framing properties. These can be easily avoided if KeY would be able to make use of the fact that instances implementing the `Uint256` interface are immutable and by tweaking the proof search strategies towards the specifics of the Solidity translations.

6 Limitations and Challenges

One of the most difficult issues to be handled by our approach is the undefined evaluation order of nested expressions in Solidity. This means the semantics of contracts with nested expressions is dependent on the compiler being used (similar to the situation in C). There are several alternatives to address this issue in our approach: (i) forbid nested expressions to be used in a Solidity contract, and to reject such contracts early on; (ii) provide compiler specific calculus rules to be chosen prior to a verification attempt, at the cost of rendering the verification result compiler specific; (iii) split the proof into one subproof for each possible evaluation order when encountering a nested expression. As all possible orders are considered, a successful verification would be meaningful independent of the used compiler. However, this can lead to rather large (number of) proofs in the presence of nested expressions; (iv) and when reaching a nested expression during symbolic execution, prove that the result is independent of the evaluation order and continue with the uniquely determined result. In our current experiment we used the first alternative, but we plan to adapt alternative (iii) or (iv) in the future.

[12] See https://github.com/raulpardo/casino-contract-java-solidity.

One of the major challenges which static verification of smart contracts faces is that of modelling the blockchain environment within which the smart contract is executed. For instance, in Solidity, one may access the current block number, timestamp of the block, and other parameters which may only be known at runtime. Our approach is to make no assumption on these values, and thus proofs must go through with the values being completely non-deterministic. In this manner, we ensure soundness but we may lose completeness when an algorithm may have been designed to use implicit constraints on these values e.g. that block numbers are strictly increasing. From our experience, few smart contracts make such assumptions, and when one wants to verify a property of such a smart contract, one can still add such assumptions explicitly.

Many of the bugs and security flaws of Solidity are due to specific decisions taken when designing the language. In the white paper [7, Chapter 4.4] Everts and Muller provide a comprehensive overview of these issues. In what follows we summarise some of these issues and explain to which extent and how we deal with them.

One class of issues is rooted in the design choice concerning the semantics of certain programming constructs in Solidity. Some examples of this are: (i) a differing semantics whether division is on literals (and precomputed by the compiler) or involves variables (evaluated at runtime), (ii) difference in the treatment of method calls depending on using an implicit or explicit `this`, i.e., the statements/expressions `this.m()` and `m()` may result in different behaviour, and (iii) usage of copy-by-reference and copy-by-value looks the same on the source code level. All of these issues are or can be easily supported by our approach at the translation level by choosing the correct Java implementation of the used Solidity construct. This is indeed possible as these differences, although invisible or surprising to the user, can be identified unambiguously by static analysis and taken care of accordingly.

Another issue is how a programming language decides to deal with integer overflow (and underflow). Solidity joins C's and Java's approach by silently overflowing. This easily leads to mistakes , as programmers often use natural or whole numbers as internal mental models. Our approach models overflow and underflow semantics faithfully and proves a program correct only if the overflow was intended and/or does not invalidate the property to be proven. KeY for Java provides also a second sound (but incomplete) approach to the same problem by enforcing to prove that no overflow happens.

7 Related Work

Although the need for formal verification, particularly compile-time static analysis techniques, for smart contracts has been highlighted various times e.g. [3,7], actual work in the domain is still sparse. Most work on static analysis techniques for smart contracts falls in one of two categories — either Lint-like syntactic analysis of code to find potential vulnerabilities like Solcheck (https://git.io/fxXeu) and Solium (https://git.io/fxXec), or semantics-based static analysis specialised to identify commonly encountered problems with smart contracts (e.g. gas leaks, reentrancy problems).

Of the latter type, one finds approaches designed for different types of vulnerabilities. Fröwis et al. [8] address smart contract control-flow mutability which is typically not desirable. OYENTE [13] is a tool which can perform reentrancy detection and other

analysis using symbolic execution. Mythril [14] uses concolic analysis, taint analysis and control-flow analysis for identifying security vulnerability, while SmartCheck (https://tool.smartdec.net) uses both Lint-like and semantic analysis to identify various vulnerabilities. Bhargavan et al. [5] transform Solidity into F* on which they perform analysis to identify vulnerable patterns. The other approaches perform their analysis at the EVM bytecode level, mainly because the control-flow analysis used typically does not use the program structure. This enables the analysis of any smart contract deployed on the Ethereum blockchain. It is worth noting that the semantics of Solidity are only informally described in the language documentation, and effectively pragmatically decided based on what the compiler does. In contrast, there are published formal semantics for EVM bytecode either through direct formalisation or via translation in [10,9,11].

Both these types of static analysis approaches have been shown to readily scale up to large smart contracts, the former because the complexity of syntactic analysis is of the order of the size of the source code , while the latter typically use overapproximations to ensure tractability. However, the downside is that neither of these approaches allow reasoning about the functional aspect of the smart contract under scrutiny, i.e. what the contract is actually trying to achieve. There is little published work towards achieving specification-specific static analysis for business logic verification of smart contracts.

Bai et al. [4] perform model checking using SPIN but perform the analysis on a model of the smart contract rather than directly on the code. Similarly, Abdellatif et al. [1] build a model not only of the smart contracts but also the underlying blockchain and miners using timed automata to enable verification. In both cases there lies a substantial gap between the actual smart contract and the model, raising questions of the faithfulness of the model with respect to the concrete code. Our approach suffers also from this issue due to the translation from Solidity to Java. However, our model is much more granular (no loss of precision), and thus the gap between our Java model and the original is much narrower.

8 Conclusions

We have presented a translation-based verifier of smart contracts using the deductive verification tool KeY. Our approach is one of the first to go beyond verifying standard sanity checks (e.g. *there are no integer over- and underflows*) and enable verification of business-logic and thus contract-specific specifications (e.g. *when a player guesses a number, the casino contract will pay her 1.8 times the bet they placed*). We implemented the translation in a tool and illustrated its use on a simple casino smart contract.

Although our results indicate that our approach is promising, our contribution uncovered new unexpected questions and challenges. The first question is how the approach fares with real-life contracts. What is promising is that the size and complexity of smart contracts is trivial compared to typical software systems and matches well with our case study. They typically run into some hundreds lines of code, and use loops sparingly due to gas concerns. This may indicate that typical smart contracts are within reach of automated verification techniques. We are currently applying our approach to a number of real-world use cases (some with known bugs) to evaluate better this claim.

Currently we depend solely on hand-waving argumentation that the semantics of Solidity and our translation match, which is a concern. However, we have to emphasise that there is no established (or otherwise) formal semantics of Solidity, with the language manual and the compiler acting as arbiters as to how constructs actually work. Until now, the only semantics available are at the EVM assembly level, making a proof of correctness of the translation impossible at present. However, translating between two structured high level formalisms— Solidity and JavaCard—we believe that the leap of faith is across a much narrower gap than using a semantics at a lower level of abstraction, and the fact that both languages have a native transaction (rollback) mechanism strengthens this point. Still, a proof of semantics-preservation is highly desirable. We are also currently investigating how to build a verification tool handling Solidity programs directly rather than via a translation.

References

1. Abdellatif, T., Brousmiche, K.: Formal verification of smart contracts based on users and blockchain behaviors models. In: NTMS'18. pp. 1–5 (2018)
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book, LNCS, vol. 10001. Springer (2016)
3. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). In: POST. LNCS, vol. 10204, pp. 164–186. Springer (2017)
4. Bai, X., Cheng, Z., Duan, Z., Hu, K.: Formal modeling and verification of smart contracts. In: ICSCA'18. pp. 322–326 (2018)
5. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Formal verification of smart contracts: Short paper. In: PLAS'16. ACM (2016)
6. Delmolino, K., Arnett, M., Kosba, A.E., Miller, A., Shi, E.: Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In: Financial Cryptography Workshop, FC'16. pp. 79–94 (2016)
7. Everts, M., Muller, F.: Will that smart contract really do what you expect it to do? (white paper) (2018), White paper
8. Fröwis, M., Böhme, R.: In code we trust? — Measuring the control flow immutability of all smart contracts deployed on ethereum. In: DPM/CBT@ESORICS'17. pp. 357–372 (2017)
9. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of ethereum smart contracts. In: POST. LNCS, vol. 10804, pp. 243–269. Springer (2018)
10. Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., Rosu, G.: KEVM: A complete semantics of the Ethereum Virtual Machine (2017), White paper
11. Hirai, Y.: Defining the ethereum virtual machine for interactive theorem provers. In: Financial Cryptography, FC'17. LNCS, vol. 10323, pp. 520–535. Springer (2017)
12. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. ACM SIGSOFT Software Eng. Notes **31**(3), 1–38 (2006)
13. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: CCS'16. pp. 254–269 (2016)
14. Mueller, B.: Smashing ethereum smart contracts for fun and real profit. In: HITB SECCONF Amsterdam (2018)
15. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System. White Paper https://bitcoin.org/bitcoin.pdf (2009)
16. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper **151**, 1–32 (2014)