Deductive Verification of SmartML Smart Contracts with KeY

Tudor Christian Balan
Department of Computer Science, Technical University of Darmstadt, Germany

Wolfram Pfeifer
KASTEL – Institute of Information Security and Dependability,
Karlsruhe Institute of Technology, Germany

Adele Veschetti
Department of Computer Science, Technical University of Darmstadt, Germany

Abstract

Unintended behavior in smart contracts can lead to major financial losses. Due to the immutable nature of blockchains, it is of utmost importance to ensure the functional correctness of smart contracts before deployment. Formal verification is a powerful technology for such critical applications, as it can show the absence of errors. Current approaches focus on verifying programs on specific blockchains, such as the Ethereum Virtual Machine (EVM). Consequently, the SmartML smart contract modeling language was developed to design smart contracts independently of any particular blockchain.

In this work, we present a novel approach for formally verifying SmartML contracts via an automatic translation to Java Card and the Java Modeling Language (JML). We extend SmartML with SmartJML, a JML-like specification language, and describe how SmartML and SmartJML can be automatically translated into Java Card and JML. With this, the established deductive verification tool KeY can be used for conducting proofs on the generated Java Card program. The faithfulness of our translation ensures that the obtained guarantees hold for the original SmartML models. In addition to the theoretical work, we provide a prototypical implementation of the automatic translation and evaluate it with a case study of an escrow.

2012 ACM Subject Classification Software and its engineering → Formal methods

Keywords and phrases Formal Verification, Deductive Verification, Smart Contract Verification

Digital Object Identifier 10.4230/OASIcs.FMBC.2026.6

Funding This work has been partially supported by the ATHENE project "Model-centric Deductive Verification of Smart Contracts".

1 Introduction

Smart contracts play an increasingly central role in financial systems, decentralized applications, and digital asset management. Their execution is typically irreversible, and even small errors in contract logic can lead to permanent loss of funds or unintended behavior. For this reason, the design of smart contracts requires a level of assurance that goes beyond traditional software testing. Yet, smart contracts combine stateful updates, external interactions, and intricate failure modes, making them difficult to analyze with conventional programming and quality-assurance techniques.

SmartML [15] was introduced to address the modeling side of this challenge. It is a domain-specific, executable modeling language with a formally defined operational semantics [13], designed to capture contract behavior at an abstract and implementation-independent level. Its key features include explicit transactional control, resource-aware operations, and algebraic data types, thereby enabling developers to express contract logic precisely and unambiguously.

With this, it is possible to write clearly structured smart contract implementations. However, the language by itself does not provide formal guarantees about correctness. For safety-critical smart contracts, such guarantees are essential.

Deductive verification is a natural fit for verifying functional correctness and transactional safety of smart contracts, as it allows one to prove the absence of entire classes of errors rather than detecting individual bugs. However, existing deductive verification tools do not support SmartML, and its distinctive constructs – most notably its explicit transactional execution model – cannot be analyzed directly within established verification frameworks.

The goal of this paper is to enable deductive verification of SmartML models using the KeY verification system. We achieve this by introducing SmartJML, a lightweight specification layer for expressing correctness properties directly in SmartML, and by defining a semantics-preserving translation from SmartML+SmartJML to Java Card annotated with JML, the input language supported by KeY. As a result, correctness proofs carried out in KeY are sound with respect to the behavior of the original SmartML contracts, including their transactional execution.

Our work builds on the previously introduced SmartML framework, which already provides the modeling language itself, its formal operational semantics, and a type-system-based analysis for safe reentrancy [15]. These aspects are assumed in the present paper and are not re-developed here. The novelty of this work lies instead in enabling deductive verification for SmartML: we introduce SmartJML as a specification layer, define a translation from SmartML+SmartJML to Java Card annotated with JML, and thereby make KeY applicable to SmartML models. In summary, the contributions of this paper are:

- we extend the existing SmartML framework with SmartJML, a JML-style specification layer for writing method contracts, invariants, and ghost variables directly at the SmartML level;
- we define a translation from SmartML+SmartJML to Java Card+JML that encodes SmartML's transactional execution model in a form suitable for deductive verification with KeY;
- we provide a tool-supported implementation of this translation pipeline, generating Java Card/JML artifacts automatically from annotated SmartML models;
- we evaluate the approach on escrow case studies and demonstrate that correctness properties of SmartML contracts can be established in KeY, including transactional reasoning.

These contributions establish a verification workflow for smart contracts, illustrated in Figure 1. Starting from a smart contract, the developer models it in SmartML and annotates it with SmartJML specifications. The translation produces a Java Card/JML representation suitable for deductive verification with KeY. A successful proof implies correctness of the original SmartML model with respect to the specified properties, and – assuming faithfulness of the model – of the intended smart contract.

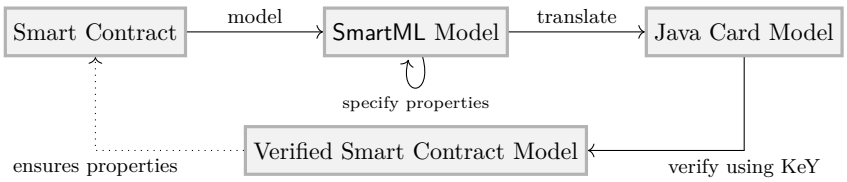

Figure 1 Workflow for smart contract verification.

The paper proceeds as follows. Section 2 provides background on SmartML and KeY. Section 3 introduces SmartJML and in Section 4 we discuss our translation to Java Card/JML. Section 5 describes our implementation, while Section 6 evaluates it on a representative case study. Finally, Section 7 discusses related work, and Section 8 concludes with future research directions.

2 Background

This section summarizes the technical background required for our verification workflow, focusing on the SmartML modeling language and the KeY deductive verification system.

2.1 SmartML

SmartML is an executable, object-oriented modeling language designed for describing smart contract behavior at a high level of abstraction. Its primary purpose is not direct deployment on a blockchain, but the precise and analyzable specification of contract logic. A SmartML program consists of contracts, resources, and algebraic data types (ADTs), all governed by a formal small-step operational semantics that models field updates, method calls, exception propagation, and control flow [15].

Syntax. An excerpt of the abstract syntax of SmartML is shown in Figure 2. We use standard metavariables: adt for datatype identifiers, Res for resource names, Exc for exception names, C, D for contract names, m, n for method and ADT-function names, f for field names, and x, v, w, g for variables. The nonterminals F, K, and M denote ADT-function declarations, constructors, and methods, respectively. Sequences are written with an overline (e.g. $\overline{f}$), and optional parts are enclosed in brackets [•]. The complete grammar of SmartML is given in [15].

P	$::= \overline{A}\ \overline{R}\ \overline{E}\ \overline{L}$	(program)
A	$::=$ datatype adt {constructor{$\overline{fn} :: \overline{adt}$} $\overline{F}$}	(datatype)
F	$::= \tau_n\ n(\overline{\tau}\ \overline{w})\ \{d\}$	(ADT function)
d	$::=$ if(e){e}else{e} | return e | $n(\overline{w})$ | switch e {case $e : d$; default : d}	(ADT expression)
R	$::=$ resource Res {constructor($\overline{f : \tau}$){this.$\overline{f} = \overline{f}$} $\overline{M}$}	(resource)
E	$::=$ exception Exc [$(\overline{x : \tau})$]	(exception)
L	$::=$ contract C [extends D] [uses $\overline{Res}$] {$\overline{f : \tau}$; K; $\overline{M}$}	(contract)
K	$::=$ constructor([$\overline{g : \tau}$,] $\overline{f : \tau}$) {[super($\overline{g}$);] this.$\overline{f} = \overline{f}$}	(constructor)
M	$::= \tau_m\ m(\overline{\tau}\ \overline{x})\ \{s\}$	(method)
s	$::=$ if(e){s}[else{s}] | while(e){s} | let $x := rhs$ in s | assert(e) | $x := rhs$ | $x := x.m(\overline{x})$ | $x.m(\overline{x})$ | return [e] | throw e | try s_0 abort {s_1} success {s_2} | $s_1; s_2$	(statement)
e	$::= v$ | e_1 op e_2 | e_1 bop e_2 | sender | amount | res	(expression)
v	$::= x$ |!v | $v.f$ | true | false | d | this$_C$	(value)
rhs	$::= e$ | new $C(\overline{v})$	(right-hand side)
op	$::= +$ | $-$ | $\times$ | $\div$	(arithmetic operator)
bop	$::= \leq$|$\geq$| && | | |$=$|$\neq$	(boolean operator)

Figure 2 Excerpt of the syntax of SmartML, including the constructs relevant to this paper.

A program consists of a sequence of ADT, resource, exception, and contract declarations. ADTs declared with datatype define immutable structured data and pure ADT functions built from conditionals, pattern matching, and function calls. Resource declarations provide an abstract notion of transferable assets; each resource defines fields, a constructor, and methods for manipulating the asset independently of any concrete blockchain.

Contracts are declared using contract C [extends D] [uses r1 ... rk] { ... } and may inherit from a single parent. A contract contains field declarations, a constructor that initializes them (and optionally invokes the superclass constructor), and a set of methods defining the contract's behavior.

Statements include the usual imperative constructs (if, while, assignments, return, throw) as well as assertions and pattern-matching catch clauses. SmartML also provides an explicit *transaction* construct: try s abort s1 success s2. This form models an external call whose effects are committed only on success; on failure, the abort block is executed and intermediate state updates are rolled back. Method calls can additionally specify resource transfers and access predefined values such as sender, amount, and res. Expressions follow standard syntax, and field access is restricted to the current object this for simplicity of the operational model.

Semantics. We do not show the semantic rules here, as they are quite extensive. The full operational semantics can be found in [15]. At a high level, a SmartML contract follows a simple state-based memory model: the persistent state consists exclusively of the contract's fields, while local variables and method parameters are transient. Field updates are the only observable state changes and may be rolled back when executed inside an aborted transaction. There is no shared mutable state or direct field access across contracts.

Overall, SmartML combines expressive modeling constructs with a rigorous operational semantics. These characteristics are crucial for the translation described in Section 4, where SmartML contracts are mapped to Java Card for verification with KeY.

2.2 KeY

To verify smart contracts, we use the deductive verification tool KeY [2]. It enables the verification of programs by checking that a Java program adheres to its Java Modeling Language (JML) specification. Beyond standard Java, KeY also supports the Java Card dialect. JML supports method contracts, class and object invariants, and auxiliary specifications such as loop invariants and specification-only fields (ghost fields) . The specification is written entirely within Java comments, so it does not affect the compilation of the Java program itself. When loading a program into KeY, proof obligations based on the program and specification are created. If these can be verified, the program adheres to its specification.

For reasoning, KeY core uses a sequent calculus for Java Dynamic Logic (JavaDL), which adds Java programs and the modalities [.] ("box") and ⟨.⟩ ("diamond") to a first-order logic (FOL) with a type hierarchy called Java first-order logic (JFOL). For a Java program p and JFOL formulae φ and ψ, $[p]\psi$ states that if p terminates, ψ holds in the final state. Dually, $\langle p\rangle\psi$ states that p terminates in a state in which ψ holds. Method contracts for total correctness can therefore be expressed as $\varphi \to \langle p\rangle\psi$, where φ denotes the contract's precondition, p the method body, and ψ the postcondition. KeY operates by building a proof tree, where nodes are sequents containing JavaDL formulas.

KeY provides different strategies and macros for automatic proof search, which are often able to find proofs without any user interaction involved. However, since finding the proof automatically is not always possible, it also grants the user the ability to manually examine and interact with the proof. Therefore, KeY provides a GUI that visualizes the past and currently loaded proofs, the current proof tree, the sequent showing the current formula which is to be proven, and the source code of the loaded file.

3 Specification with SmartJML

This section introduces SmartJML, our JML-inspired specification layer for SmartML. SmartJML provides a set of JML-style specification constructs that can be embedded directly into SmartML programs. In the following, we introduce these constructs step by step using small, focused examples. SmartJML is written as comments inside the SmartML code.

Method contracts. The basic building block of SmartJML is a method contract with pre- and postconditions, analogously to JML:

```
1  contract Counter {
2      int val;
3
4      /*@ requires val >= 0;
5        @ ensures  val == \old(val) + 1;
6        @ assignable val; @*/
7      inc() { val = val + 1; }
8  }
```

SmartML + SmartJML —

Here the requires clause constrains the state before the method call, while the ensures clause describes the state afterwards. Furthermore, the assignable clause records the program locations that can change during method execution and the expression \old(val) refers to the value of val in the pre-state, as in JML.

Access to contract state and parameters. SmartJML expressions range over the fields and parameters of the contract, while local variables and temporary computations are not visible in specifications. This ensures that specifications refer only to the observable state of the contract. For example, a method that transfers funds and records the recipient can be specified as follows:

```
1  /*@ requires amount > 0 && r != this;
2    @ ensures  balance == \old(balance) - amount && lastRecipient == r;
3    @ assignable balance, lastRecipient; @*/
4  pay(address r) { /* SmartML code manipulating balance and lastRecipient */ }
```

SmartML + SmartJML —

Invariants. SmartJML allows the specification of invariants, which describe properties of contract fields that must be maintained across method executions. For example:

```
1  //@ ghost int length;
2  //@ public invariant (length > 0);
```

SmartML + SmartJML —

The invariant semantics follows KeY: invariants are assumed in the pre-state of a method call and must be re-established in the post-state; invariants of other objects can be referenced via `\invariant_for(o)`.

Moreover, expressions in SmartJML are fully compatible with JML expressions [9], allowing for formal reasoning about contract behavior within a familiar framework. SmartJML also supports specification-only variables (ghost variables).

3.1 Transactional Specifications

A key feature of SmartML is the explicit notion of transactions. Statements of the form `try ... abort ... success ...` execute in a transactional context and can either *commit* or *abort*. As a rollback occurs in case a transaction *aborts*, the specification has to differentiate between the behavior of a transactional *commit* and a transactional *abort*. To be able to address both cases separately in the specification, SmartJML introduces two modifiers for method contracts.

`transaction_success`. The modifier transaction_success marks a contract as describing the post-state in the case where the surrounding transaction commits. Consider a simplified deposit operation:

```
1  /*@ transaction_success
2    @ ensures (balance == \old(balance) + amount) && (\result == true);
3    @ assignable balance;
4    @*/
5  bool deposit() {
6      try BankCoin$(amount).transfer(sender, this);
7      abort { return false; }
8      success { balance = balance + amount; return true; }
9  }
```

SmartML + SmartJML

The ensures clause talks only about the state when the transaction *commits* and the success branch is reached. If the external transfer fails and control flows to abort, this contract does not apply.

`transaction_abort`. The transaction_abort modifier allows one to specify the expected state of the contract when a transaction is aborted, complementing the usual postconditions for successful execution. It is particularly useful for reasoning about operations that may fail and need to roll back any state changes. For example:

```
1  /*@ transaction_abort
2    @ ensures \result == false;
3    @ assignable \nothing; @*/
4  bool deposit() {
5      try BankCoin$(amount).transfer(sender, this);
6      abort { return false; }
7      success { balance = balance + amount; return true; }
8  }
```

SmartML + SmartJML

In practice, one may either give two separate specification cases – one for the commit outcome (transaction_success) and one for the rollback outcome (transaction_abort) – or restrict attention to the commit case when an abort restores the pre-state and no additional guarantees are required beyond signaling failure. During translation, both modifiers are preserved and mapped to corresponding JML annotations in the generated Java Card program.

4 Translation to Java Card/JML

KeY reasons about Java programs annotated with JML and provides dedicated support for the Java Card dialect; it has no native notion of SmartML contracts, resources, or transactions. We therefore translate each SmartML construct into a Java Card/JML counterpart that simulates its semantics. The translation is defined on top of the formal operational semantics of SmartML, and we only sketch its main ideas here.

From a semantic perspective, a SmartML program is executed according to the operational semantics defined in [15]. A method contract in SmartJML therefore represents a partial correctness specification over these semantics: a precondition P and postcondition Q constrain the set of reachable configurations before and after evaluating a SmartML method. In particular, SmartML's transactional execution model induces two kinds of postconditions, depending on whether a transaction commits or aborts. This distinction is reflected explicitly in SmartJML.

The goal of our translation is to ensure that such specifications are preserved under translation: iff a SmartML program satisfies a SmartJML contract with respect to its formal semantics, then the generated Java Card program satisfies the corresponding JML contract with respect to the Java/JML semantics implemented in KeY. Formally, for a method m, we can write

$$C \models_{\mathcal{S}} P_\rho \to \langle m \rangle Q_\rho \quad \Longleftrightarrow \quad T(C) \models_{\text{KeY}} P_\rho \to \langle m \rangle Q_\rho,$$

where $\rho \in \{\mathsf{success}, \mathsf{abort}\}$ captures the transactional outcome and T denotes our translation. Therefore, in this section, we outline the design choices and requirements that make this preservation feasible.

4.1 Contracts

Every SmartML contract becomes a Java class with the same fields and a single constructor. The methods and corresponding JML contracts are obtained from the SmartML methods and SmartJML contract by a straightforward syntactic translation; we reuse JML syntax wherever possible so that the user sees the same specification in both representations. For instance, the Counter contract from above is translated to a Java class:

```
1  public class SL_Counter {
2    private int val;
3    /*@ requires val >= 0;
4      @ ensures  val == \old(val) + 1;
5      @ assignable val; @*/
6    public void inc() {
7      val = val + 1;
8    }
9  }
```

Java Card + JML —

Execution Context: `sender`, `amount`, and `res`. In SmartML, methods implicitly have access to the caller address sender, the transferred amount amount, and the corresponding resource type res. Java methods, in contrast, have only explicit parameters. The translation therefore introduces additional parameters for these context variables. For example, the SmartML method pay is translated as follows:

```
pay() {
    // SmartML code
}
```
— SmartML + SmartJML —

```
public void pay(Address sender, int amount, Resource res) {
    // Java code
}
```
— Java Card + JML —

Here, Address and Resource are Java classes generated from the corresponding SmartML declarations. This makes all contextual information explicit for verification.

Transactions. The try ... abort ... success ... construct is one of the most characteristic features of SmartML. The following example is given:

```
try  externalCall();
abort   { S_abort }
success { S_success }
```
— SmartML + SmartJML —

When externalCall () succeeds, the success branch executes and its effects are committed; otherwise, control flows to the abort branch and all persistent state changes since the beginning of the transaction are reverted.

In the generated Java Card program we realize this behavior using Java Card's transactional memory mechanism [11]. Verifying transactions in this way is only possible with Java Card, not Java. Conceptually, the translation wraps the body of the transaction with calls to the Java Card API:

```
JCSystem.beginTransaction();
try {
  externalCall();
  // S_success translated to Java
  JCSystem.commitTransaction();
} catch (Exception e) {
  JCSystem.abortTransaction();
  // S_abort translated to Java
}
```
— Java Card + JML —

KeY supports reasoning about Java Card transactions by internally maintaining a separate heap for rollback [2]. This allows us to prove the transactional properties that were specified in SmartJML using transaction_success and transaction_abort. We do not allow for the translation of nested transactions within a method as they are not supported by Java Card.

Assertions in SmartML also have a rollback mechanism in case of failure. However, as Java Card does not allow for nested transactions within a method, implementing assertions as transactions is not feasible. We still allow assertions to be used as the first and last line of a method, as is commonly done for smart contracts. These are translated into further pre- and postconditions, respectively.

4.2 Resources

A resource in SmartML is a construct representing a finite currency that can be traded between contracts. While syntactically similar to contracts, once a resource is declared, a singleton instance of the resource is automatically created. To ensure in Java that only one instance of each resource exists, we translate all resource declarations into classes which adhere to the singleton design pattern. That is, each resource is translated to a Java class with a static field whose type is the resource class, a single private constructor, and a static method get() which checks if the field has been instantiated, instantiates it if necessary, and returns the instance.

4.3 Algebraic Data Types

SmartML supports algebraic data types (ADTs), which allows users to define their own data types and perform pattern matching on them. These ADTs are immutable by nature and are created and expanded in a recursive manner. Consequently, functions that operate on ADTs frequently use recursion. ADTs are translated into immutable Java classes with one constructor per SmartML constructor and an additional tag field used to simulate pattern matching. The ADT Java classes are marked as pure in JML. The full encoding is omitted for brevity, as it is straightforward and does not affect the presentation of our main results.

In the current implementation, ADT specifications cannot yet be expressed purely at the SmartML level and instead must refer to the translated Java encoding for technical reasons. As future work, we plan to extend SmartJML with complete ADT support, enabling specifications written in terms of SmartML ADT structure and pattern matching to be translated into corresponding JML specification functions over the generated Java classes. This will likely require additional proof support (e.g., automatically generated lemmas) to bridge the abstract ADT view and its concrete Java representation.

5 Implementation

The implementation follows the theoretical translation principles introduced in the previous section and produces a Java Card program annotated with JML. This program can then be loaded directly into KeY for verification. We also provide a repository containing our implementation[1].

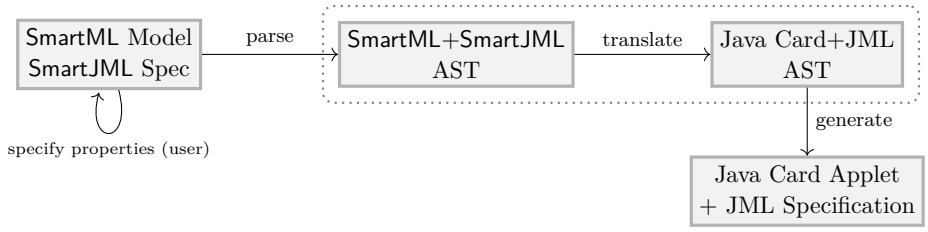

Figure 3 Translator workflow.

The translator is implemented in Java and follows a structure similar to that of a source-to-source compiler: it takes a SmartML program as input, parses it into a parse tree which our translator turns into an abstract syntax tree (AST), transforms this AST into a JML-annotated Java Card AST, and finally generates Java Card source code compatible with KeY. The workflow of our translator is illustrated in Figure 3.

Our translator also translates SmartJML specifications to JML. This feature does not yet cover all possible SmartJML constructs, which is planned as future work. While not a fundamental limitation of the approach, some auxiliary specifications – such as loop invariants – currently need to be written directly in the generated Java Card program. In practice, this affects only a small portion of the specifications: in our case study, loop invariants were required only for recursive operations over ADTs, while the remaining contracts were translated automatically without manual intervention.

[1] https://github.com/TudorBalan/SmartJML

In the implementation, our translator uses ANTLR [12] and JMLParser[2], which extends JavaParser [14] to include functionality for parsing and generating JML code. The parser is generated using ANTLR and the code generation is almost completely handled by JMLParser. This only leaves us having to implement the translation of the two ASTs by hand, for which both ANTLR and JMLParser again provide useful functionality. The choice of our tools led to a major reduction in required development time. Furthermore, following best practices for building a source-to-source compiler, such as the usage of the visitor pattern and designing the translator modularly, has led to a robust code base that is easily extendable for when SmartML is expanded in the future.

6 Case Study

To illustrate our verification workflow, we consider a practical example: a SmartML escrow contract. Escrow contracts are commonly used in online marketplaces, where a third party holds the buyer's funds until the promised goods are delivered. Our case study shows how to model the contract in SmartML, specify its properties in SmartJML, translate it to Java Card, and verify it using KeY.

Contract Requirements. The escrow contract should satisfy the following functional requirements: (i) Receive a deposit from the buyer; (ii) Hold the funds securely until delivery; (iii) Allow the buyer to confirm receipt of the goods; (iv) Release the funds to the seller upon confirmation; (v) Allow the buyer to cancel the transaction before confirmation.

The contract maintains three fields: buyer, seller, and funds. The funds field also acts as an implicit flag for ongoing transactions: funds $= 0$ indicates no active transaction, and funds > 0 indicates an ongoing transaction.

SmartML Model and SmartJML Specification. Listing 1 shows the SmartML escrow contract together with its SmartJML specifications. Each method contract uses the transaction_success keyword, since all operations are executed only upon successful transaction completion. In cases where the preconditions are violated, one could also define a dual contract based on transaction_abort. These additional contracts are omitted due to space constraints.

Properties to Verify. We aim to verify two main properties:

1. **Functional correctness:** Each method performs the intended operations. This is ensured by the SmartJML method contracts.
2. **Escrow transaction integrity:** The funds field correctly indicates the transaction state: funds $= 0$ when no transaction is active and funds > 0 during a transaction. Formally, for operations constructor CC, `startTransaction` ST, `confirmArrival` CA, and `cancelTransaction` CT, a simplified proof obligation would be:

$$C \models_{\mathcal{S}} (\{funds = 0\} \rightarrow \langle ST \rangle \{funds > 0\}_{\tau} \wedge (\{funds > 0\} \rightarrow \langle CA \rangle \{funds = 0\}_{\tau}) \\ \wedge (\{funds > 0\} \rightarrow \langle CT \rangle \{funds = 0\}_{\tau}) \wedge (\{true\} \rightarrow \langle CC \rangle \{funds = 0\}_{\tau})).$$

For modularity, this is split by KeY into several individual proof obligations. When all of them are proven, we obtain the correctness guarantee that funds works as specified and that we can rely on it as an implicit flag for ongoing transactions. After translation, the automatic proof strategy in KeY is sufficient to verify all method contracts without manual intervention, provided no ADTs are used.

[2] JMLParser repository on GitHub: `https://github.com/jmltoolkit/jmlparser`

Listing 1 Escrow contract.

```
contract Escrow uses CoinInfo {
    address seller;
    address buyer;
    int funds;

    /*@ ensures funds == 0; @*/
    constructor() { funds = 0; }

    /*@ transaction_success
      @ ensures funds == amount && seller == _seller && buyer == sender;
      @ assignable funds, seller, buyer; @*/
    startTransaction(address _seller) {
        assert((amount > 0) && (funds == 0));
        try CoinInfo$(amount).transfer(sender, this);
        abort {}
        success { funds = amount; seller = _seller; buyer = sender; }
    }

    /*@ transaction_success
      @ ensures funds == 0;
      @ assignable funds; @*/
    confirmArrival() {
        assert(sender == buyer && funds > 0);
        try CoinInfo$funds.transfer(this, seller);
        abort {}
        success { funds = 0; }
    }

    /*@ transaction_success
      @ ensures funds == 0;
      @ assignable funds; @*/
    cancelTransaction() {
        assert(sender == buyer && funds > 0);
        try CoinInfo$funds.transfer(this, buyer);
        abort {}
        success { funds = 0; }
    }
}
```

SmartML + SmartJML —

Extending to Multiple Transactions with ADTs. To demonstrate verification in the presence of recursive data structures, we extend the escrow contract from handling a single active transaction to supporting multiple concurrent transactions. Instead of storing one triple `(buyer, seller, funds)` in the contract state, the extended model maintains a collection of escrow entries represented as a list-like ADT. Each entry captures the participants and locked funds of one transaction, together with an identifier that uniquely distinguishes it from other active transactions. Contract operations are generalized accordingly: initiating an escrow adds a new entry to the collection, while confirming delivery or canceling a transaction updates the state of the corresponding entry and releases the associated funds.

Listing 2 shows the ADT definition together with SmartJML specifications. The length ghost field and class invariant track the depth of each node, while the measured_by clause in recursive methods ensures termination. Methods such as getIndex allow the contract to locate specific transactions in the list.

After the translation to Java Card and JML, the model is successfully verified in KeY. A quantitative evaluation of the proven method contracts from Listing 1 and Listing 2 can be seen in Table 1. All contracts of the escrow model in Listing 1 are proven fully

Listing 2 ADT of a list. As discussed in Section 4.3, ADT specifications currently refer to the translated Java encoding rather than the original SmartML ADT syntax.

```
datatype EscrowList {
    //@ public ghost \bigint length;
    /*@ public invariant (0 <= length) && (sl_isNil <==> length == 0) &&
      @                 (sl_isNil <==> tail == null) &&
      @                 (tail != null ==> (length == tail.length + 1 &&
      @                 \invariant_for(tail))); @*/

    constructor { nil :: EscrowList | cons(int transactionID, address seller, address buyer,
                                           int funds, EscrowList tail) :: EscrowList }

    /*@ requires \invariant_for(list);
      @ ensures \invariant_for(list) && (list.sl_isNil ==> \result == -1) &&
      @         (!list.sl_isNil && list.transactionID == n ==> \result == 0) &&
      @         (!list.sl_isNil && list.transactionID != n &&
      @         getIndex(list.tail, n) == -1 ==> \result == -1) &&
      @         (!list.sl_isNil && list.transactionID != n &&
      @         getIndex(list.tail, n) >= 0 ==> \result == getIndex(list.tail, n) + 1) &&
      @         (-1 <= \result <= list.length) && (\result == getIndex(list, n));
      @ measured_by list.length; @*/
    int getIndex(EscrowList list, int n) { ... }
}
```

SmartML + SmartJML —

automatically by KeY's proof strategy in under one second. For the extended version using recursive ADTs, the proofs become more complex and require a small number of manual proof steps. This increase in complexity is reflected in the larger proof trees and longer average step times shown in the table. From a user perspective, this indicates that simple SmartML contracts without recursive data structures can typically be verified with little knowledge of the underlying proof calculus, as the automatically generated proof obligations are handled by KeY's automated strategies. More complex models involving ADTs and recursion, however, may require additional user interaction and a deeper understanding of the KeY proof framework.

Table 1 Quantitative evaluation of case study method contracts.

Method	Proof steps	Interactive steps	Auto. time (ms)	Avg. step time (ms)
startTransaction	292	0	901	3.096
confirmArrival	254	0	785	3.102
cancelTransaction	254	0	833	3.292
getIndex	1,883	8	71,154	37.807

The integration of deductive verification into the SmartML language enables the verification of a wide range of smart contract properties. Many important safety guarantees already follow from the design of SmartML itself. For instance, the language provides protection against reentrancy attacks [15] and mitigates unchecked external calls through its transactional construct `try ... abort ... success ...`. Because these mechanisms make the control flow of external interactions explicit, many correctness and security properties can be expressed directly as simple pre- and postconditions over the contract state. In practice, as illustrated by the escrow case study, such specifications can often be verified automatically by KeY without requiring deep knowledge of the underlying proof calculus.

In addition, KeY allows reasoning about method parameters and arithmetic operations. This makes it possible to verify that smart contracts enforce appropriate input validation through preconditions and that arithmetic operations do not cause integer overflow or underflow. In practice, such properties can often be expressed using simple constraints in the method specifications. For more complex properties, auxiliary specifications may be required to guide the proof.

To put our verification workflow into context, Figure 4 illustrates which classes of vulnerabilities from the widely recognized OWASP Smart Contract Top 10[3] can be addressed by our approach.

SC01:2025	Access Control Vulnerabilities
SC02:2025	Price Oracle Manipulation
SC03:2025	Logic Errors
SC04:2025	Lack of Input Validation
SC05:2025	Reentrancy Attacks
SC06:2025	Unchecked External Calls
SC07:2025	Flash Loan Attacks
SC08:2025	Integer Overflow and Underflow
SC09:2025	Insecure Randomness
SC10:2025	Denial of Service (DoS) Attacks

Figure 4 OWASP Smart Contract Top 10 (2025) Coverage.

7 Related Work

Due to the safety and security of smart contracts being of utmost importance, given the financial incentives involved, numerous approaches have been proposed for the purpose of verifying the behavior of smart contracts and preventing attacks. The use of KeY for the deductive verification of smart contracts is discussed in [7]. This work shows that multiple classes of correctness properties are suitable for verification with KeY. The authors also present an approach for the deductive verification of Hyperledger Fabric smart contracts with KeY in [6]. They achieve this by extending KeY with functionality for Fabric ledger implementations, which differs from our translation approach. Another verification methodology that is quite similar to ours is proposed in [8], which focuses on verifying smart contracts written within the EVM environment and uses Dafny instead of KeY.

A similar approach to the translator which we have built is seen in Javadity [4], a translator for Solidity programs to Java including JML for the purpose of verifying these programs using KeY. Solidity is a programming language designed for the development of smart contracts which run on the Ethereum Virtual Machine (EVM). Consequently, Javadity is only capable of verifying smart contracts which have been written in Solidity and are therefore designed to run on the EVM. SolidiKeY [3] is a verification tool for smart contracts written for Solidity and based on KeY, which is also limited to the verification of smart contracts written in Solidity. The process of translating SmartML and SmartJML to Java

[3] https://owasp.org/www-project-smart-contract-top-10

Card and JML differs notably from previous approaches that translate Solidity programs to Java for verification with KeY. One key difference is that SmartML now provides a dedicated specification language (SmartJML), allowing contracts to be annotated with verification properties directly at the modeling level before translation. Furthermore, SmartML includes language constructs that do not have direct counterparts in Solidity, such as the explicit transactional statement try ... abort ... success ..., which models rollback semantics for external calls. The translation therefore needs to encode these semantics explicitly using Java Card's transactional memory mechanism. In addition, SmartML follows a simplified state model in which the persistent state consists only of contract fields and no shared mutable state across contracts. These characteristics lead to different translation challenges compared to Solidity-based approaches.

Another approach of verifying smart contracts is through the use of model checking, as in [5] the authors propose a formal method to construct models for smart contracts in ProMeLa. These models can then be verified using the SPIN model checker. Similarly, in [1] the authors employ a statistical model checking approach in which the smart contracts as well as the users and the decentralized ledger are all represented as timed automata. Furthermore, static analysis techniques are used for smart contracts, as for example seen with Oyente [10]. Oyente is a tool used for bug finding that has the capacity to perform reentrancy detection as well as other analyses. However, this method relies on symbolic execution which makes it unable to guarantee the absence of false negatives. This approach is also not compositional, thereby disabling verification that is independent of the external contract's behavior. Consequently, Oyente is incapable of blocking cross-contract reentrancy, thereby hindering the ability to protect against cross-contract reentrancy attacks. Since SmartML is blockchain-agnostic, our approach is not tied to the EVM and supports expressive first-order specifications beyond typical finite-state model-checking workflows.

8 Conclusion

In this paper we proposed a verification workflow to deductively verify SmartML smart contract models using the well-established verification tool KeY. We did this by developing a translation from SmartML smart contract models to Java Card to allow the use of KeY for SmartML models. We found that the translation cannot be fully equivalent in every case due to assertions reverting fields in SmartML. Therefore, we are able to translate a subset of SmartML that does not allow nested transactions within a method and limits assertions to the first and last line of a method. We also extended SmartML with SmartJML to allow for specifications to be written within SmartML programs. Furthermore, we developed a tool that automates the translation process from SmartML and SmartJML to Java Card and JML. Some SmartML programs still require manual translation of some SmartJML expressions, especially when working with algebraic data types. Lastly, we have also shown our verification workflow in action through the context of a case study.

The main advantage of our verification workflow is that it allows smart contracts to be deductively verified, demonstrating the absence of errors, not just their presence, in addition to the safety and security guarantees already provided by SmartML. Meanwhile, its main drawback is that it becomes increasingly complex to prove properties about a program the larger it gets. Especially when working with algebraic data types, the effort required to verify a smart contract model increases substantially due to KeY's increased proof complexity when dealing with recursion. Nonetheless, having the ability to formally verify smart contract models, thereby increasing the functional correctness and security of critical applications such as smart contracts, is highly useful.

As of now, the translator only supports a subset of SmartJML specification. Expanding the translator's SmartJML to JML translation would further increase the accessibility and usefulness of writing specification in SmartJML.

References

1 Tesnim Abdellatif and Kei-Léo Brousmiche. Formal verification of smart contracts based on users and blockchain behaviors models. In *NTMS*, pages 1–5. IEEE, 2018. doi:10.1109/NTMS.2018.8328737.

2 Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016. doi:10.1007/978-3-319-49812-6.

3 Wolfgang Ahrendt and Richard Bubel. Functional verification of smart contracts via strong data integrity. In *ISoLA (3)*, volume 12478 of *Lecture Notes in Computer Science*, pages 9–24. Springer, 2020. doi:10.1007/978-3-030-61467-6_2.

4 Wolfgang Ahrendt, Richard Bubel, Joshua Ellul, Gordon J. Pace, Raúl Pardo, Vincent Rebiscoul, and Gerardo Schneider. Verification of smart contract business logic - exploiting a java source code verifier. In *FSEN*, volume 11761 of *Lecture Notes in Computer Science*, pages 228–243. Springer, 2019. doi:10.1007/978-3-030-31517-7_16.

5 Xiaomin Bai, Zijing Cheng, Zhangbo Duan, and Kai Hu. Formal modeling and verification of smart contracts. In *ICSCA*, pages 322–326. ACM, 2018. doi:10.1145/3185089.3185138.

6 Bernhard Beckert, Mihai Herda, Michael Kirsten, and Jonas Schiffl. Formal specification and verification of hyperledger fabric chaincode. In *3rd Symposium on Distributed Ledger Technology (SDLT-2018) co-located with ICFEM*, pages 44–48, 2018.

7 Bernhard Beckert, Jonas Schiffl, and Mattias Ulbrich. Smart contracts: Application scenarios for deductive program verification. In *FM Workshops (1)*, volume 12232 of *Lecture Notes in Computer Science*, pages 293–298. Springer, 2019. doi:10.1007/978-3-030-54994-7_21.

8 Franck Cassez, Joanne Fuller, and Horacio Mijail Anton Quiles. Deductive verification of smart contracts with dafny. In *FMICS*, volume 13487 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 2022. doi:10.1007/978-3-031-15008-1_5.

9 Gary T Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M Zimmerman, et al. Java modeling language (JML) reference manual, 2008.

10 Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *CCS*, pages 254–269. ACM, 2016. doi:10.1145/2976749.2978309.

11 Marcus Oestreicher. Transactions in java card. In *ACSAC*, pages 291–298. IEEE Computer Society, 1999. doi:10.1109/CSAC.1999.816039.

12 Terence John Parr and Russell W. Quong. ANTLR: A predicated- *LL(k)* parser generator. *Softw. Pract. Exp.*, 25(7):789–810, 1995. doi:10.1002/spe.4380250705.

13 Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.*, 60-61:17–139, 2004.

14 Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. *JavaParser: Visited*. Leanpub, 2023.

15 Adele Veschetti, Richard Bubel, and Reiner Hähnle. A formal modeling language for smart contracts. In *SEFM*, volume 15280 of *Lecture Notes in Computer Science*, pages 89–106. Springer, 2024. doi:10.1007/978-3-031-77382-2_6.

A Appendix

Listing 3 Escrow contract translation.

```
class Escrow extends Address {
    private Address seller;
    private Address buyer;
    private int funds;

    /*@ normal_behavior
      @ ensures funds == 0;              @*/
    public Escrow(){
        this.funds = 0;
    }

    /*@ normal_behavior
      @ requires (JCSystem.getTransactionDepth() == 0) && ((amount > 0) && (funds == 0));
      @ ensures ((funds > 0) && (funds == amount) && (seller == _seller) &&
      @         (buyer == sender)) || ((funds == \old(funds)) &&
      @         (seller == \old(seller)) && (buyer == \old(buyer)));
      @ assignable funds, seller, buyer; @*/
    public void startTransaction(int amount, CoinInfo res, Address sender, Address _seller) {
        try {
            JCSystem.beginTransaction();
            CoinInfo.transfer(sender, amount, this);
            JCSystem.commitTransaction();
            funds = amount;
            seller = _seller;
            buyer = sender;
        } catch (Exception e) {
            JCSystem.abortTransaction();
        }
    }

    /*@ normal_behavior
      @ requires (JCSystem.getTransactionDepth() == 0) && ((sender == buyer) && (funds > 0));
      @ ensures (funds == 0) || (funds == \old(funds));
      @ assignable funds;                @*/
    public void confirmArrival(int amount, CoinInfo res, Address sender) {
        try {
            JCSystem.beginTransaction();
            CoinInfo.transfer(this, amount, seller);
            JCSystem.commitTransaction();
            funds = 0;
        } catch (Exception e) {
            JCSystem.abortTransaction();
        }
    }

    /*@ normal_behavior
      @ requires (JCSystem.getTransactionDepth() == 0) && ((sender == buyer) && (funds > 0));
      @ ensures (funds == 0) || (funds == \old(funds));
      @ assignable funds;                @*/
    public void cancelTransaction(int amount, CoinInfo res, Address sender) {
        try {
            JCSystem.beginTransaction();
            CoinInfo.transfer(this, amount, buyer);
            JCSystem.commitTransaction();
            funds = 0;
        } catch (Exception e) {
            JCSystem.abortTransaction();
        }
    }
}
```

— Java Card + JML —