---

title: Formal verification of smart contracts description: An overview of formal verification for Ethereum smart contracts lang: en

---

Smart contracts are making it possible to create decentralized, trustless, and robust applications that introduce new use-cases and unlock value for users. Because smart contracts handle large amounts of value, security is a critical consideration for developers.

Formal verification is one of the recommended techniques for improvingsmart contract security. Formal verification, which uses formal methods for specifying, designing, and verifying programs, has been used for years to ensure correctness of critical hardware and software systems.

When implemented in smart contracts, formal verification can prove that a contract's business logic meets a predefined specification. Compared to other methods for assessing the correctness of contract code, such as testing, formal verification gives stronger guarantees that a smart contract is functionally correct.

What is formal verification? {#what-is-formal-verification}

Formal verification refers to the process of evaluating the correctness of a system with respect to a formal specification. In simpler terms, formal verification allows us to check if the behavior of a system satisfies some requirements (i.e., it does what we want).

Expected behaviors of the system (a smart contract in this case) are described using formal modeling, while specification languages enable the creation of formal properties. Formal verification techniques can then verify that the implementation of a contract complies with its specification and derive mathematical proof of the former's correctness. When a contract satisfies its specification, it is described as “functionally correct”, “correct by design”, or “correct by construction”.

What is a formal model? {#what-is-a-formal-model}

In computer science, a formal model is a mathematical description of a computational process. Programs are abstracted into mathematical functions (equations), with the model describing how outputs to functions are computed given an input.

Formal models provide a level of abstraction over which analysis of a program's behavior can be evaluated. The existence of formal models allows for the creation of a *formal specification*, which describes desired properties of the model in question.

Different techniques are used for modeling smart contracts for formal verification. For example, some models are used to reason about the high-level behavior of a smart contract. These modeling techniques apply a black-box view to smart contracts, viewing them as systems that accept inputs and execute computation based on those inputs.

High-level models focus on the relationship between smart contracts and external agents, such as externally owned accounts (EOAs), contract accounts, and the blockchain environment. Such models are useful for defining properties that specify how a contract should behave in response to certain user interactions.

Conversely, other formal models focus on the low-level behavior of a smart contract. While high-level models can help with reasoning about a contract's functionality, they may fail capture details about the internal workings of the implementation. Low-level models apply a white-box view to program analysis and rely on lower-level representations of smart contract applications, such as program traces and control flow graphs, to reason about properties relevant to a contract's execution.

Low-level models are considered ideal since they represent the actual execution of a smart contract in Ethereum's execution environment (i.e., the EVM). Low-level modeling techniques are especially useful in establishing critical safety properties in smart contracts and detecting potential vulnerabilities.

What is a formal specification? {#what-is-a-formal-specification}

A specification is simply a technical requirement that a particular system must satisfy. In programming, specifications represent general ideas about a program's execution (i.e., what the program should do).

In the context of smart contracts, formal specifications refer to *properties*—formal descriptions of the requirements that a contract must satisfy. Such properties are described as "invariants" and represent logical assertions about a contract's execution that must remain true under every possible circumstance, without any exceptions.

Thus, we can think of a formal specification as a collection of statements written in a formal language that describe the intended execution of a smart contract. Specifications cover a contract's properties and define how the contract should behave in different circumstances. The purpose of formal verification is to determine if a smart contract possesses these properties (invariants) and that these properties are not violated during execution.

Formal specifications are critical in developing secure implementations of smart contracts. Contracts that fail to implement invariants or have their properties violated during execution are prone to vulnerabilities that can harm functionality or cause malicious exploits.

Types of formal specifications for smart contracts {#formal-specifications-for-smart-contracts}

Formal specifications enable mathematical reasoning about the correctness of program execution. As with formal models, formal specifications can capture either high-level properties or the low-level behavior of a contract implementation.

Formal specifications are derived using elements of program logic, which allow for formal reasoning about the properties of a program. A program logic has formal rules that express (in mathematical language) the expected behavior of a program. Various program logics are used in creating formal specifications, including reachability logic, temporal logic, and Hoare logic.

Formal specifications for smart contracts can be classified broadly as either **high-level** or **low-level** specifications. Regardless of what category a specification belongs to, it must adequately and unambiguously describe the property of the system under analysis.

High-level specifications {#high-level-specifications}

As the name suggests, a high-level specification (also called a "model-oriented specification") describes the high-level behavior of a program. High-level specifications model a smart contract as a finite state machine (FSM), which can transition between states by performing operations, with temporal logic used to define formal properties for the FSM model.

Temporal logics are "rules for reasoning about propositions qualified in terms of time (e.g., "I am *always* hungry" or "I will *eventually* be hungry")." When applied to formal verification, temporal logics are used to state assertions about the correct behavior of systems modeled as state-machines. Specifically, a temporal logic describes the future states a smart contract can be in and how it transitions between states.

High-level specifications generally capture two critical temporal properties for smart contracts: **safety** and **liveness**. Safety properties represent the idea that “nothing bad ever happens” and usually express invariance. A safety property may define general software requirements, such as freedom from deadlock, or express domain-specific properties for contracts (e.g., invariants on access control for functions, admissible values of state variables, or conditions for token transfers).

Take for example this safety requirement which covers conditions for using the `transfer()` or `transferFrom()` in ERC-20 token contracts: *“A sender’s balance is never lower than requested amount of tokens to be sent..”* This natural-language description of a contract invariant can be translated into a formal (mathematical) specification, which can then be rigorously checked for validity.

Liveness properties assert that “something eventually good happens” and concern a contract’s ability to progress through different states. An example of a liveness property is “liquidity”, which refers to a contract's ability to transfer its balances to users on request. If this property is violated, users would be unable to withdraw assets stored in the contract, like what happened with the Parity wallet incident.

Low-level specifications {#low-level-specifications}

High-level specifications take as a starting point a finite-state model of a contract and define desired properties of this model.

In contrast, low-level specifications (also called "property-oriented specifications") often model programs (smart contracts) as systems comprising a collection of mathematical functions and describe the correct behavior of such systems.

In simpler terms, low-level specifications analyze *program traces* and attempt to define properties of a smart contract over these traces. Traces refer to sequences of function executions that alter the state of a smart contract; hence, low-level specifications help specify requirements for a contract's internal execution.

Low-level formal specifications can be given as either Hoare-style properties or invariants on execution paths.

Hoare-style properties {#hoare-style-properties}

Hoare Logic provides a set of formal rules for reasoning about the correctness of programs, including smart contracts. A Hoare-style property is represented by a Hoare triple $\{P\}c\{Q\}$, where c is a program and P and Q are predicates on the state of the c (i.e., the program), formally described as *preconditions* and *postconditions*, respectively.

A precondition is a predicate describing the conditions required for the correct execution of a function; users calling into the contract must satisfy this requirement. A postcondition is a predicate describing the condition that a function establishes if correctly executed; users can expect this condition to be true after calling into the function. An *invariant* in Hoare logic is a predicate that is preserved by execution of a function (i.e., it doesn't change).

Hoare-style specifications can guarantee either *partial correctness* or *total correctness*. The implementation of a contract function is "partially correct" if the precondition holds true before the function is executed, and if execution terminates, the postcondition is also true. Proof of total correctness is obtained if a precondition is true before the function executes, the execution is guaranteed to terminate and when it does, the postcondition holds true.

Obtaining proof of total correctness is difficult since some executions may delay before terminating, or never terminate at all. That said, the question of whether execution terminates is arguably a moot point since Ethereum's gas mechanism prevents infinite program loops (execution terminates either successfully or ends due to 'out-of-gas' error).

Smart contract specifications created using Hoare logic will have preconditions, postconditions, and invariants defined for the execution of functions and loops in a contract. Preconditions often include the possibility of erroneous inputs to a function, with postconditions describing the expected response to such inputs (e.g., throwing a specific exception). In this manner Hoare-style properties are effective for assuring correctness of contract implementtions.

Many formal verification frameworks use Hoare-style specifications for proving semantic correctness of functions. It is also possible to add Hoare-style properties (as assertions) directly to contract code by using the `require` and `assert` statements in Solidity.

`require` statements express a precondition or invariant and are often used to validate user inputs, while `assert` captures a postcondition necessary for safety. For instance, proper access control for functions (an example of a safety property) can be achieved using `require` as a precondition check on the identity of the calling account. Similarly, an invariant on permissible values of state variables in a contract (e.g., total number of tokens in circulation) can be protected from violation by using `assert` to confirm the contract's state after function execution.

Trace-level properties {#trace-level-properties}

Trace-based specifications describe operations that transition a contract between different states and the relationships between these operations. As explained earlier, traces are sequences of operations that alter the state of a contract in a particular way.

This approach relies on model of smart contracts as state-transition systems with some predefined states (described by state variables) along with a set of predefined transitions (described by contract functions). Furthermore, a control flow graph (CFG), which is a graphical representation of a program's execution flow, is often used for describing operational semantics of a contract. Here, each trace represented as a path on the control flow graph.

Primarily, trace-level specifications are used to reason about patterns of internal execution in smart contracts. By creating trace-level specifications, we assert the admissible execution paths (i.e., state transitions) for a smart contract. Using techniques, such as symbolic execution, we can formally verify that execution never follows a path not defined in the formal

model.

Let's use an example of a DAO contract that has some publicly accessible functions to describe trace-level properties. Here, we assume the DAO contract allows users to perform the following operations:

- Deposit funds
- Vote on a proposal after depositing funds
- Claim a refund if they don't vote on a proposal

Example trace-level properties could be *"users that do not deposit funds cannot vote on a proposal"* or *"users that do not vote on a proposal should always be able to claim a refund"*. Both properties assert preferred sequences of execution (voting cannot happen *before* depositing funds and claiming a refunds cannot happen *after* voting on a proposal).

Techniques for formal verification of smart contracts {#formal-verification-techniques}

Model checking {#model-checking}

Model checking is a formal verification technique in which an algorithm checks a formal model of a smart contract against its specification. In model checking smart contracts are often represented as state-transition systems, while properties on permissible contract states are defined using temporal logic.

Model checking requires creating an abstract mathematical representation of a system (i.e., a contract) and expressing properties of this system using formulas rooted in propositional logic. This simplifies the task of the model-checking algorithm, namely to prove that a mathematical model satisfies a given logical formula.

Model checking in formal verification is primarily used to evaluate temporal properties that describe the behavior of a contract over time. Temporal properties for smart contracts include *safety* and *liveness*, which we explained earlier.

For example, a security property related to access control (e.g., *Only the owner of the contract can call* `selfdestruct`) can be written in formal logic. Thereafter, the model-checking algorithm can verify if the contract satisfies this formal specification.

Model checking uses state space exploration, which involves constructing all possible states of a smart contract and attempting to find reachable states that result in property violations. However, this can lead to an infinite number of states (known as the "state explosion problem"), hence model checkers rely on abstraction techniques to make efficient analysis of smart contracts possible.

Theorem proving {#theorem-proving}

Theorem proving is a method of mathematically reasoning about the correctness of programs, including smart contracts. It involves transforming the model of a contract's system and its specifications into mathematical formulas (logic statements).

The objective of theorem proving is to verify logical equivalence between these statements. "Logical equivalence" (also called "logical bi-implication") is a type of relationship between two statements such that the first statement is true *if and only if* the second statement is true.

The required relationship (logical equivalence) between statements about a contract's model and its property is formulated as a provable statement (called a theorem). Using a formal system of inference, the automated theorem prover can verify the theorem's validity. In other words, a theorem prover can conclusively prove a smart contract's model precisely matches its specifications.

While model checking models contracts as transition systems with finite states, theorem proving can handle analysis of infinite-state systems. However, this means an automated theorem prover cannot always know if a logic problem is "decidable" or not.

As a result, human assistance is often required to guide the theorem prover in deriving correctness proofs. The use of

human effort in theorem proving makes it more expensive to use than model checking, which is fully automated.

Symbolic execution {#symbolic-execution}

Symbolic execution is a method of analyzing a smart contract by executing functions using *symbolic values* (e.g., `x > 5`) instead of *concrete values* (e.g., `x == 5`). As a formal verification technique, symbolic execution is used to formally reason about trace-level properties in a contract's code.

Symbolic execution represents an execution trace as a mathematical formula over symbolic input values, otherwise called a *path predicate*. An SMT solver is used to check if a path predicate is "satisfiable" (i.e., there exists a value that can satisfy the formula). If a vulnerable path is satisfiable, the SMT solver will generate a concrete value that triggers steers execution toward that path.

Suppose a smart contract's function takes as input a `uint` value (`x`) and reverts when `x` is greater than `5` and also lower than `10`. Finding a value for `x` that triggers the error using a normal testing procedure would require running through dozens of test cases (or more) without the assurance of actually finding an error-triggering input.

Conversely, a symbolic execution tool would execute the function with the symbolic value `X > 5 ∧ X < 10` (i.e., `x` is greater than 5 AND `x` is less than 10). The associated path predicate `x = X > 5 ∧ X < 10` would then be given to an SMT solver to solve. If a particular value satisfies the formula `x = X > 5 ∧ X < 10`, the SMT solver will calculate it—for example, the solver might produce `7` as a value for `x`.

Because symbolic execution relies on inputs to a program, and the set of inputs to explore all reachable states is potentially infinite, it is still a form of testing. However, as shown in the example, symbolic execution is more efficient than regular testing for finding inputs that trigger property violations.

Moreover, symbolic execution produces fewer false positives than other property-based techniques (e.g., fuzzing) that randomly generate inputs to a function. If an error state is triggered during symbolic execution, then it is possible to generate a concrete value that triggers the error and reproduce the issue.

Symbolic execution can also provide some degree of mathematical proof of correctness. Consider the following example of a contract function with overflow protection:

``` function safe_add(uint x, uint y) returns(uint z){

z = x + y; require(z>=x); require(z>=y);

return z; ```

An execution trace that results in an integer overflow would need to satisfy the formula: `z = x + y AND (z >= x) AND (z=>y) AND (z < x OR z < y)` Such a formula is unlikely to be solved, hence it serves a mathematical proof that the function `safe_add` never overflows.

## Why use formal verification for smart contracts? {#benefits-of-formal-verification}

### Need for reliability {#need-for-reliability}

Formal verification is used to assess the correctness of safety-critical systems whose failure can have devastating consequences, such as death, injury, or financial ruin. Smart contracts are high-value applications controlling enormous amounts of value, and simple errors in design can lead to irrecoverable losses for users. Formally verifying a contract before deployment, however, can increase guarantees that it will perform as expected once running on the blockchain.

Reliability is a highly desired quality in any smart contract, especially because code deployed in the Ethereum Virtual Machine (EVM) is typically immutable. With post-launch upgrades not readily accessible, the need to guarantee reliability of contracts makes formal verification necessary. Formal verification is able to detect tricky issues, such as integer underflows and overflow, re-entrancy, and poor gas optimizations, which may slip past auditors and testers.

### Prove functional correctness {#prove-functional-correctness}
```

Program testing is the most common method of proving that a smart contract satisfies some requirements. This involves executing a contract with a sample of the data it is expected to handle and analyzing its behavior. If the contract returns the expected results for the sample data, then developers have objective proof of its correctness.

However, this approach cannot prove correct execution for input values that are not part of the sample. Therefore, testing a contract may help detect bugs (i.e., if some code paths fail to return desired results during execution), but **it cannot conclusively prove the absence of bugs**.

Conversely, formal verification can formally prove that a smart contract satisfies requirements for an infinite range of executions *without* running the contract at all. This requires creating a formal specification that precisely describes correct contract behaviors and developing a formal (mathematical) model of the contract's system. Then we can follow a formal proof procedure to check for consistency between the contract's model and its specification.

With formal verification, the question of verifying if a contract's business logic satisfies requirements is a mathematical proposition that can be proved or disproved. By formally proving a proposition, we can verify an infinite number of test cases with a finite number of steps. In this manner formal verification has better prospects of proving a contract is functionally correct with respect to a specification.

Ideal verification targets {#ideal-verification-targets}

A verification target describes the system to be formally verified. Formal verification is best used in "embedded systems" (small, simple pieces of software that form part of a larger system). They are also ideal for specialized domains that have few rules, as this makes it easier to modify tools for verifying domain-specific properties.

Smart contracts—at least, to some extent—fulfill both requirements. For example, the small size of Ethereum contracts makes them amenable to formal verification. Similarly, the EVM follows simple rules, which makes specifying and verifying semantic properties for programs running in the EVM easier.

Faster development cycle {#faster-development-cycle}

Formal verification techniques, such as model checking and symbolic execution, are generally more efficient than regular analysis of smart contract code (performed during testing or auditing). This is because formal verification relies on symbolic values to test assertions ("what if a user tries to withdraw *n* ether?") unlike testing which uses concrete values ("what if a user tries to withdraw 5 ether?").

Symbolic input variables can cover multiple classes of concrete values, so formal verification approaches promise more code coverage in a shorter time frame. When used effectively, formal verification can accelerate the development cycle for developers.

Formal verification also improves the process of building decentralized applications (dapps) by reducing costly design errors. Upgrading contracts (where possible) to fix vulnerabilities requires extensive rewriting of codebases and more effort spent on development. Formal verification can detect many errors in contract implementations that may slip past testers and auditors and provides ample opportunity to fix those issues before deploying a contract.

Drawbacks of formal verification {#drawbacks-of-formal-verification}

Cost of manual labor {#cost-of-manual-labor}

Formal verification, especially semi-automated verification in which a human guides the prover to derive correctness proofs, requires considerable manual labor. Moreover, creating formal specification is a complex activity that demands a high level of skill.

These factors (effort and skill) makes formal verification more demanding and expensive compared to the usual methods of assessing correctness in contracts, such as testing and audits. Nevertheless, paying the cost for a full verification audit is practical, given the cost of errors in smart contract implementations.

False negatives {#false-negatives}

Formal verification can only check if the execution of the smart contract matches the formal specification. As such, it is important to make sure the specification properly describes the expected behaviors of a smart contract.

If specifications are poorly written, violations of properties—which point to vulnerable executions—cannot be detected by the formal verification audit. In this case, a developer might erroneously assume that the contract is bug-free.

Performance issues {#performance-issues}

Formal verification runs into a number of performance issues. For instance, state and path explosion problems encountered during model checking and symbolic checking, respectively, can affect verification procedures. Also, formal verification tools often use SMT solvers and other constraint solvers in their underlying layer, and these solvers rely on computationally intensive procedures.

Also, it is not always possible for program verifiers to determine if a property (described as a logical formula) can be satisfied or not (the "decidability problem") because a program might never terminate. As such, it may be impossible to prove some properties for a contract even if it's well-specified.

Formal verification tools for Ethereum smart contracts {#formal-verification-tools}

Specification languages for creating formal specifications {#specification-languages}

Act: *Act allows specification of storage updates, pre/post conditions and contract invariants. Its tool suite also has proof backends able to prove many properties via Coq, SMT solvers, or hevm.*

- GitHub
- Documentation

Scribble - *Scribble transforms code annotations in the Scribble specification language into concrete assertions that check the specification.*

- Documentation

Dafny - *Dafny is a verification-ready programming language that relies on high-level annotations to reason about and prove correctness of code.*

- GitHub

Program verifiers for checking correctness {#program-verifiers}

Certora Prover - *Certora Prover is an automatic formal verification tool for checking code correctness in smart contracts. Specifications are written in CVL (Certora Verification Language), with property violations detected using a combination of static analysis and constraint-solving.*

- Website
- Documentation

Solidity SMTChecker - *Solidity's SMTChecker is a built-in model checker based on SMT (Satisfiability Modulo Theories) and Horn solving. It confirms if a contract's source code matches specifications during compilation and statically checks for violations of safety properties.*

- GitHub

solc-verify - *solc-verify is an extended version of the Solidity compiler that can perform automated formal verification on Solidity code using annotations and modular program verification.*

- GitHub

KEVM - *KEVM is a formal semantics of the Ethereum Virtual Machine (EVM) written in the K framework. KEVM is*

executable and can prove certain property-related assertions using reachability logic.

- GitHub
- Documentation

Logical frameworks for theorem proving {#theorem-provers}

Isabelle - *Isabelle/HOL is a proof assistant that allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas. The main application is the formalization of mathematical proofs and in particular formal verification, which includes proving the correctness of computer hardware or software and proving properties of computer languages and protocols.*

- GitHub
- Documentation

Coq - *Coq is an interactive theorem prover that lets you define programs using theorems and interactively generate machine-checked proofs of correctness.*

- GitHub
- Documentation

Symbolic execution-based tools for detecting vulnerable patterns in smart contracts {#symbolic-execution-tools}

Manticore - *A tool for analyzing EVM bytecode analysis tool based on symbolic execution*

- GitHub
- Documentation

hevm - *hevm is a symbolic execution engine and equivalence checker for EVM bytecode.*

- GitHub

Mythril - *A symbolic execution tool for detecting vulnerabilities in Ethereum smart contracts*

- GitHub
- Documentation

Further reading {#further-reading}

- How Formal Verification of Smart Contracts Works
- How Formal Verification Can Ensure Flawless Smart Contracts
- An Overview of Formal Verification Projects in the Ethereum Ecosystem
- End-to-End Formal Verification of Ethereum 2.0 Deposit Smart Contract
- Formally Verifying The World's Most Popular Smart Contract
- SMTChecker and Formal Verification