Pruvendo Formal Verification Approach

Description of formal verification approach

Smart contracts, which handle large amounts of money and are part of the open Internet, are frequently targeted by hackers with advanced attack techniques (such as the Lazarus group, which is responsible for up to 35% of the overall stolen funds in web3). For example, the Pruvendo team discovered a project with a hidden vulnerability. Despite having only $3 in liquidity and no prior announcement, the project was targeted by automated hacking systems.

Smart contracts differ from traditional software by being grounded in decentralization principles and not providing total system control. This often leads to the inability to resolve the consequences of incorrect program operation, resulting in unpleasant effects, up to the freezing of funds.

Industry experts estimate that the web3 sector suffers substantial financial losses, amounting to billions of dollars annually, as a result of errors in smart contracts. Pruvendo is aware of over 170 cases with a total damage exceeding 6 billion dollars.

Considering the circumstances, traditional quality assurance methods (QA), including architectural methods, best programming practices, code review, multi-level testing, post-production monitoring, etc., remain necessary but not sufficient to ensure the required code quality.

Traditionally, the smart contract industry relies on audits to tackle this issue. However, the effectiveness of this approach is not consistently satisfactory. For instance, in the midst of the intense work phase on the FreeTon project (now known as Everscale), there were instances where exceptionally skilled audit teams couldn't identify a single shared bug in one smart contract (**list of found vulnerabilities of every team was completely different to each other**), highlighting the limitations of manual audits.

The **fundamental solution to the problem is formal verification**, namely, a mathematical (in the form of a set of theorems and lemmas) method of proving program correctness (or rather, its compliance with specifications).

This approach was developed about half a century ago but remained too complex and expensive for practical use, having a niche solely for high-budget projects (such as the Paris metro, the Hercules military transport aircraft, etc.).

However, in recent years, a number of projects have been carried out aiming to adapt the aforementioned concept for the mass market.

Pruvendo is one of the few companies in the formal verification of software market that has managed to adapt the most comprehensive and complete approach for the broader industry application - **deductive verification**, which is completely strict in the mathematical sense of the term.

The result of formal verification is formal proof. Proof is a program that can be independently verified using a process known as Typechecking. This process is performed in a distinct system - Proof Assistant. This is the reason why the results of formal verification do not require additional validation. Proof shows 100% correspondence of implementation to specification (in the mathematical sense of the term). Thus, any incorrect behavior of the program can be caused only by an incorrect specification. To reduce the risk of incorrect specification, Pruvendo has developed its own technology for specification development (it's based on a multistep approach that allows it to deep down gradually thus observing the whole project from multiple points of view).

The outcome of the formal verification is a comprehensive **range of services that results in**:

- **Proof** of minimal chance of errors (bug-free guarantees are impossible because of risks of mistakes in specifications)
- Or a **detailed list of encountered mistakes** along with their degree of significance

Thus, the formal verification is effectively splitted into two big tasks:

- Formal specification
- Formal verification itself

An approach for both tasks will be considered below.

Formal specification approach

Introduction

When formal verification is being discussed it's important to understand that it's an approach to mathematically prove not the *correctness* of the particular software program (*correctness* is just a common sense term that means nothing in a formal world (roughly speaking, it's the same as *do the right job*, but what *right* means?)), but rather it's about proving the equivalence (or, stricter speaking, isomorphism) between the **implementation** being verified and some entity called **formal specification**.

There are many formal specification approaches such as TLA+ , Z notation etc. sometimes based on rather advanced technologies such as λ-calculus, however, in the opinion of

Pruvendo, all of them have the following critical drawbacks that prevents them from using in the industrial formal verification:

- The customer (usually, software architect) need to dive too deep in the world of the new concepts that prevents him from the adequate assessing the provided specification, thus increasing risk of errors from the specifier
- There is a high risk of missing some important specification terms, thus increasing the possibility that some important bugs remain unfound

The drawbacks mentioned above significantly decrease an ability of the formal verification to become a magic bullet for the mission-critical software.

Pruvendo suggests its own originally developed approach that intents to:

- Provide step-to-step move from the business-level specification to the formal one to let the customer to stop at the stage of comfort and understanding
- Help to the specifier not to forget anything, thus getting the full specification rather than one that misses the critical statements

Mathematically, the suggested approach has the strong mathematical base on Category theory, toposes and monads, however, its understanding is not required for reading the present document.

The present document is intended to help the reader to understand the whole concepts as well as to get the particular specification up to the comfort stage.

Top-level specification

Business-level specification

The specification creation starts from the business-level specification. It's just a text document that describes:

- Purpose of the project
- Key concepts
- The detailed description of the project behavior

This document is supposed to be understandable by any customer and must be approved by him to ensure everything is caught by the specifier correctly.

Stage 1. Scenarios

The key idea of the presented paradigm is the scenario-based approach. While the scenario is a logically bound useful interaction with the software (or, in some cases, elements of such an interaction), the whole specification is considered as a set of scenarios (rather independent to each other).

The task of choosing if a particular logically bound construction is a scenario or not is a sole decision of the specifier (can be roughly considered with a decision to put some software code into a separate file or not).

An example of a scenario system is provided in the Appendix A, that should help to understand what scenario means.

High-level specification

High-level specification is intended to provide step-to-step formal specification without being tightly bound to the implementation. Thus, it's important to mention that some entities defined in the high-level specification **don't directly correspond to the implementation entities**. Such a collision is to be resolved at the low level of the specification.

Technological notes

Currently, the high-level specification technologically is based on draw.io/diagrams.net with some additional Kotlin-based proprietary tools to perform some transformations of the diagrams as well as some extra auxiliary activity.

In the near future *Pruvendo* plans to move to their own toolchain.

Stage 2. Output

The first stage of each scenario is to identify the outputs. Indeed, nobody from software development follows Porthos, a character from the novel of Dumas, who fought just for fighting. The reason (output) of each scenario is the only justification for its existence. The following graphical elements present here:

Rectangle	Description of the *outcome* in a natural language. Different rectangles stay for different outcomes
Parentheses	Combine the different *rectangles* into the same *outcome*
Out	The terminal element of the *Output*. Its meaning is under discussion, may be removed in future
Arrows	The connecting lines between the core part of the *Output* and *Out*. May be removed in future

Stage 3. Input

When the *Output* is defined, it's time to define *Input*, so the set of *Actors* that initiate the scenario. The following types of *Actors* may exist:

- *Human* - it's an external actor that commonly acts from direct instructions of human being or a software that pretends to be made from protein

    - Humans can be different (say, *Owner* is a different *Human* than just a *User*)
- *Cloud* - it's an external actor that presents an oracle - off-chain software module that somehow manages the workflow for smart contracts
- *Triangle*(autostart) - it's an on-chain actor (the higher scenario) that triggers the action. This kind of *Actors* can indicate the upper scenario (or scenarios) that may trigger it

Each *Actor* can trigger one or more actions defined by the corresponding arrows linked with the action name.

Stage 4. Body

Stage 4a. States I

To be sure that nothing is forgotten the concept of states has been introduced. In the most simple case there are just two states: *Nothing* (aka 'before action') and *Created* (aka 'after action'). In case of more complex scenarios the list of states can be more thorough. As an example, consider a software bug lifetime, say, in Jira: it can contain dozens of different states (say, *Created*, *Accepted*, *Evaluated*, …).

In the same way, a number of different states can be defined here. However, it's important to state that it's just a helping stage that may be skipped.

Stage 4b. Main body

When the *Output* (what is intended to reach) and *Input* (what initiated the scenario) it's time to define the stuff in the middle, called *Body*. Normally, it consists of six types of elements:

- Set of rectangles (*decision matrix*):
    - Set of exception conditions in a natural form
    - *Otherwise*, as a positive path
- Framed rectangles - we call them *masks*, empty as this stage, will be explained below
- Named rectangles - at this stage just a name of the particular transformation of data, without details. We call them *transformation elements*
- Bold rectangles - *subscenarios*, to be defined later in the same way as regular scenarios
- Crossed circles - we call them *mask cancellation*
- Connecting arrows

It's important to mention that at this step we define the scenario specification in the form of its skeleton, without any details.

Stage 5. Details

While the previous stages define the skeleton of the scenario, the present stage finalizes high-level specification by setting data types, data objects, their relations and transformations.

Roughly speaking, it's a full specification not bound to the implementation.

Stage 5a. Types and objects

As in many classic programming languages, the *types* and *objects* are introduced here. Types can be primitive, sets, and custom.

The following primitive types are defined:

Type	Meaning
A	Address. It's important to mention that this type is not obliged to correspond to, say, *address* type in Solidity, but just represents some entity that is unique for any corresponding object
B	Boolean. Just a regular logical type with two objects: *true* and *false*
N	Unsigned number. It's important to mention that this type corresponds to the mathematical $ℵ \cup \{0\}$ extended natural set. All the upper boundaries typically are not subject to high-level specification
S	Scenario type. Corresponds to the set of scenarios and subscenarios
UUID	Similar to *A*, can be used when the specifier wants to distinguish entities of a different nature
M	Abstraction for message, heterogeneous type, that has the following fields: ● r - Recipient (as A) ● a - Amount(Value) (as N) ● d - Data as an arbitrary nullable type

Like in most programming languages, *sets* are unordered homogeneous collections of the objects of particular type. It is worth mentioning that currently no inheritance is supported, so the strict type equivalence is assumed.

Custom types are usually heterogeneous structures that unite objects of different nature and type.

There are some built-in objects:

Object	Meaning
l	It's an object of the *custom* type *Ledger*. Basically, it's a root object of the whole blockchain state, where only objects important for the scenario being considered are identified
s	It's an *A*-typed object that sends the initial message (aka *msg.sender*)

ss	It's a *S*-typed object that invokes the scenario being considered (actual for subscenarios only)

The system of types and objects for the particular scenario is represented by a mixed graph, that is supposed to be intuitively clear for understanding, with the following assumptions:

- Types are represented by ellipses
- Objects are represented by the named connection arrays
- Single objects (not sets) are represented by thin arrays
- Sets are represented by thick arrays

Some built-in functions and shortcuts are also introduced for some objects:

- For any *A*-object - *b* (or *balance*) (`<no parameters>`) - balance of the objects in terms of the native currency (such as *ETH* or *TRX*)
- For any *A*-object that corresponds to non fungible token (such as *ERC20* or *TRC20*):
    - It can be directly accessed by its name enclosed in apostrophes (such as *'USDT'*)
    - It has the following functions:
        - *b* (or *balance*) (`<owner:A>`) - token balance of the `owner`
        - *a* (or *allowance*) (`<allower:A, allowee:A>`) - allowance of the `allower` to the `allowee`

Important!!! All the types and objects being described in the present section may or may not correspond to the types and objects of the implementation. While it's recommended to somehow follow the implementation to make it more understandable and ease the creation of low-level stuff, the final decision is up to the decision of the specifier.

Stage 5b. States II

This auxiliary step helps to transform the purely descriptive Stage I step into the detailed one. Its sole purpose is to provide better understanding of the specification.

Technically the step being described is a copy of Stage I with the detailed description (in terms of relations between objects) what each particular stage means.

Stage 5c. Details

When the types and objects are defined (see Stage 5a) it's time to complete high-level specification by adding object workflow, restrictions and transformations into the scenario skeleton acquired at the Stages 2-4.

Let's start with the *Input* part:

- If the particular *Actor* is restricted it's accompanied with a formula that shows if the *Actor* is allowed to perform the specific action or not
- Input arguments:

- All the input arguments are enclosed by cylinders, where, upon their first appearance, their type is directly provided in the corresponding callouts
- If the arguments come from the upper scenarios, they initially appear in the arrow preceding the *Actor* or the whole *Input* part
- Otherwise, the initially appear at the action arrow

The *Body* part is the most tricky one:

- All the input arguments (including newly created ones) follow the same rule as above
- *Decision matrices* are extended with formulae that provide a logical value if the particular condition is met or not
- *Masks* are extended with the list of entities that **can** be modified. The following rules are in place:
  - If any item of a structure (object if the *Custom* type) is modified, the structure itself is considered as not modified
- *Transformation elements* are extended with the list of transformation rules. Among the common sense rules (plain formulae) the following syntax is used:
  - For sets, the following notations are used:
    - *s*+=*x* - the set is rather unchanged, but one more element is added
    - *s*-=*x* - the set is rather unchanged, but one element is removed
  - For primitive types, the following notations are used:
    - *a*+=*x* - *a* is increased by *x* as a result of the transformation
    - *a*-=*x* - *a* is decreased by *x* as a result of the transformation
  - For all other cases:
    - Object without apostrophe - before transformation
    - Object with apostrophe - after transformation

For the *Output* part, normally nothing is changed, otherwise the same notations are used.

Stage 5d. Invariants

While graphical specification is considered as more friendly, understandably and full than traditional ones, the verifiers work with statements, not the lines.

So the final step is to convert the pictures into a set of mathematical expressions. Surprisingly, this activity is almost mechanical and will eventually be performed fully automatically. To understand this step one must:

- Read thoroughly the previous sections
- Understand the Hoare logic. It's simple:
  - *{A}B{C}* - must be read as:
    - If predicate *A*:
    - Then, after transformation *B*:
    - *C*
  - $\frac{A}{B} \Leftrightarrow (A \Leftrightarrow B)$
  - Basically, it just defines the state before and after some particular transformation

- To handle the case '*nothing else is changed*' the special character is introduced (*crossed up arrow*), that means that a particular predicate does not depend on the particular object or its descendants (in terms of structures)

Upon the completion of this step, high-level specification is fully defined, however, for the formal verification, a mapping of the specification entities (types, objects, statements) to the implementation ones is required, however, it's considered in a section below.

Stage 6: Low-level specification

Unlike high-level specification that is somehow agnostic about the implementation, low-level specification puts all the statements aligned with implementation entities (such as contracts, structures and variables) . Such an activity leads to generation of three following tables for each scenario (the first one can be single for a group of scenarios).

Axioms

This table provides a set of statements that are accepted without proof. Typically, they are either related to:

- Language-specific behavior
- Restrictions from outer systems (such as web3 applications)
- Features of the external smart contracts not to be verified (such as, say, external staking system)

The resulting table for axioms has three columns:

- *Axiom ID* - two letters (that are supposed to describe a domain for the axiom), dot and number (such as *EI.2*)
- *Human description* - description of the axiom in English
- *Formal description* - mathematical formal description, where:
    - Hoare logic is used, as above
    - Unlike in high-level specification, implementation entities are used rather than abstract concept entities

Also, some global (inside the range of the particular scenario or the set of scenarios) first-order logic predicates can be used in the third column leaving the former ones empty.

Mapping

Moving to low-level specification it's important to map all the high-level entities to the implementation-specific ones. The corresponding table is auxiliary, but helps to understand the correspondence between high and low levels. The mapping table has the following columns:

- Name of the high-level entity
- Name of the corresponding low-level entity
- Comments

It's important to highlight that this section is intended exclusively for better understanding and not to be used explicitly by verifiers.

Low-level invariants

It's the main section of the low-level specification, intended for direct usage of verifiers and serving as an ultimate result of specifiers. Technically, it's a table with the following columns:

- *Statement ID* - mapped from the high-level specification
- *Human description* - mapped from the high-level specification
- *Formal description* - can be either:
    - empty and gray, if the corresponding high-level statement is not applicable anymore
    - transformed from high-level representation into low-level one using mapping discussed above[1]

As above:

- Hoare logic is used
- Some scenario-wide (or more local) predicates can be used in separate lines

Stage 7: Conversion to Coq

When all the statements are fully specified, the transformation of them into Coq statements is nothing more but a routine activity, such as changing of ∀ to `forall`, while Hoare logic is fully supported by Coq. Currently this activity is manual, but it's planned to be fully automated[2].

Upon completion of this stage, the goal of the formal specification is fully completed - it is:

- Understandable by any PM or team lead without need to learn something new, until the last stages, that can be fully automated
- Full, as a detailed multi-step process brings a probability of missing something to a minimal value
- Correct, as the developers are able to understand it, as mentioned above

As a summary, the described process virtually eliminates the Achilles' heel of the formal verification - pool form specification.

[1] Currently this process is manual, the partial or full automation is planned in foreseeable future
[2] Automation of *Stage7* is in the roadmap

Formal verification approach

Conversion to Ursus

Most of the smart contract languages, such as Solidity, are imperative, while the proof assistants, such as Coq, are based on declarative languages, such as OCaml. To resolve this issue a special intermediate language called *Ursus* has been developed. Technically it's a DSL over Gallina (OCaml dialect used by Coq) that it's syntactically very close to Solidity. So, just see the following example:

Solidity code	Ursus code

`function _deleteUpdateRequest(uint64 updateId, uint8 index) inline private {`
`  m_updateRequestsMask &= ~(uint32(1) << index);`
`  delete m_updateRequests[updateId];`
`  }`
	`#[private, nonpayable]`
	`Ursus Definition _deleteUpdateRequest (updateId : uint64) (index : uint8): UExpression PhantomType false .`
	`{`
	`  ::// m_updateRequestsMask &= ( ~ ( (uint32(1) << {index}))) .`
	`  ::// m_updateRequests[updateId] ->delete |`
	`}`
	`return.`
	`Defined.`
	`Sync.`

It's easy to see that while the syntaxes are different, they can be mutually mapped from Solidity to Ursus and vice versa.

Currently the following translators to Ursus are implemented:

- TVM Solidity (Everscale, GOSH)
- Classic Solidity (Ethereum, Tron, other EVM-compatible chains)
- Rust (MultiversX)
- FunC (TonCoin)

Ursus also can be used as a primary language for development, with later translation to Solidity. Such an approach makes a formal verification easier, as Ursus prevents a developer from introducing some code patterns that bring some complication for the verifications.

The detailed Ursus specification and source code of translators can be provided by request.

Evals&Execs

While imperative code is successfully turned into declarative one, the state can be wrapped into a state monad, where each method is considered as a combination of two functions that correspondingly return:

- *Eval* - return value of the method
- *Exec* - the modified state

Pruvendo has developed a tool called *Generator* that performs automatic generation of *evals* and *execs* for the most practical cases.

Verification

All the previous steps can be considered as preparation for this one. Indeed, by this stage the two main artifacts are available:

- Formal specification as a set of Coq statements
- Implementation as a set of Coq functions

So, at this point nothing prevents the verifiers from proving that the implementation corresponds to the specification that is the ultimate goal of the formal verification.

To simplify this process a number of so-called Coq *tactics* (proof steps) has been developed and currently this process is semi-automated, while still requires involvement of high-skill professionals.

By the end of this stage the set of the Coq statements (treated as lemmas or theorems) is either:

- *Proved* - that means a full correspondence of the implementation to the specification, in this case the result of the verification process is positive
- *Can not be proven* - in this case the obstacle that prevents the statements from being proven is to be identified and discussed with the development team. Finally, the obstacle is treated either as:
    - Specification bug - specification is to be changed
    - Minor note - specification is to be changed, while this note must be reflected in the final report
    - Major note (bug) - the implementation must be fixed, otherwise the positive verdict is not granted

As a result of the described process the probability of the undiscovered critical bugs becomes extremely low that allows to claim a system that successfully passed the formal verification process as **reliable**.