Generating Formal Specifications for Smart Contracts from Textual Descriptions in Natural Language

Gabriel N. Leite[1]**, Filipe Arruda**[1]**, Augusto Sampaio**[1]

[1] Centro de Informática (CIn) – Universidade Federal de Pernambuco (UFPE)
Caixa Postal 7851 – 50732-970 – Recife – PE – Brazil

{gnl2, fmca, acas}@cin.ufpe.br

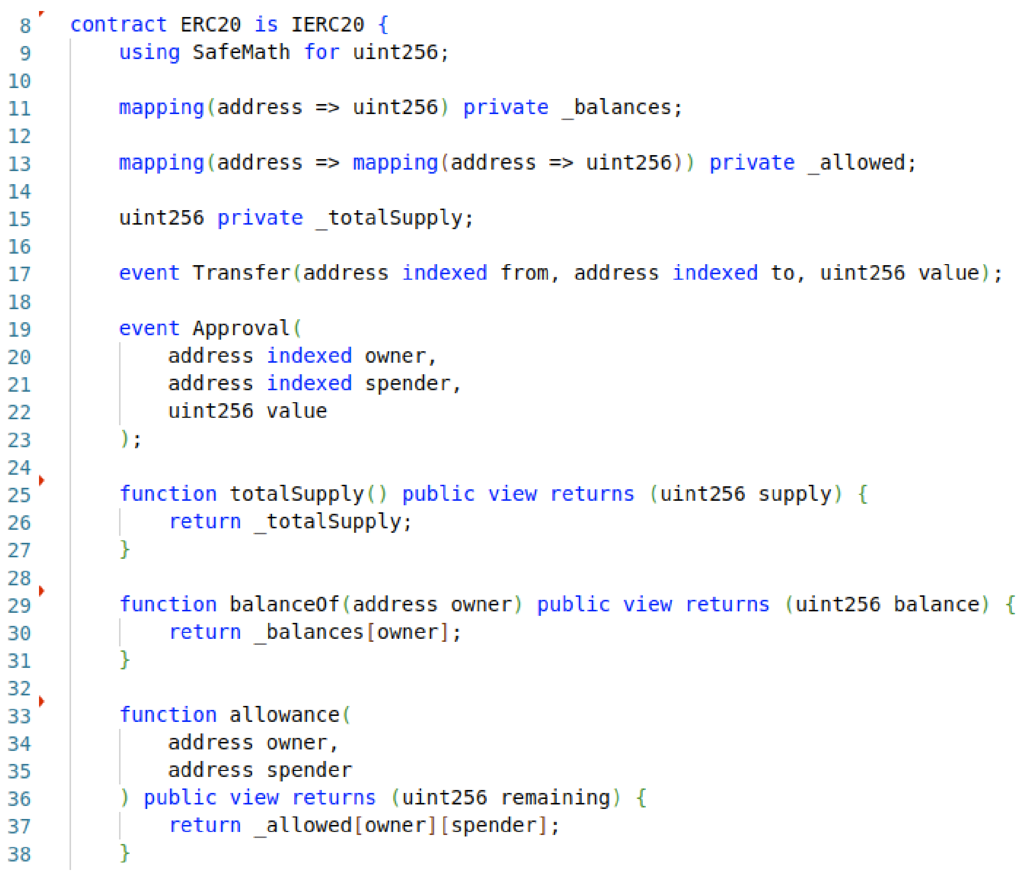
Abstract. *The increasing adoption of smart contracts in decentralized finance (DeFi) and in other areas has led to a growing need for robust and error-free code. This paper addresses this challenge by introducing a grammar-based approach for generating formal specifications from textual descriptions in natural language, specifically post-conditions for ERC20 functions of Solidity smart contracts. Particularly, our approach allows developers to transform natural language descriptions into formal specifications, and vice versa (bidirectional). Thus, developers can automatically derive postconditions from comments to verify conformance, or even generate textual descriptions for complex formal annotations in existing smart contracts to improve legibility.*

Resumo. *A crescente adoção de contratos inteligentes em finanças descentralizadas (DeFi) e em outras áreas levou a uma necessidade crescente de código robusto e livre de erros. Este artigo aborda esse desafio introduzindo uma abordagem baseada em gramática para gerar especificações formais a partir de descrições textuais em linguagem natural, especificamente pós-condições para funções ERC20 de contratos inteligentes escritos em Solidity. Particularmente, nossa abordagem permite que os desenvolvedores transformem descrições de linguagem natural em especificações formais e vice-versa (bidirecional). Assim, os desenvolvedores podem derivar automaticamente pós-condições de comentários para verificar a conformidade ou até mesmo gerar descrições textuais para anotações formais complexas em contratos inteligentes existentes para melhorar a legibilidade.*

1. Introduction

Writing smart contracts in Solidity is a challenge. It involves the application of unconventional methods of programming paradigms, due to the inherent characteristics of blockchain-based program execution. Furthermore, bugs in deployed contracts can have serious consequences, due to the immediate coupling of contract code and financial values. Therefore, it is beneficial to have a solid foundation of established and proven design and code patterns that facilitate the process of writing functional and error-free code [8].

Solidity is a Turing-complete high-level programming language with a similar syntax to JavaScript, being statically typed, supporting inheritance and polymorphism, as well as libraries and complex user-defined types [3]. Using the Solidity language, developers can write self-executing smart contracts and deploy them on Ethereum to create decentralized applications. Ethereum is a blockchain platform that provides tools for developers to create decentralized applications that, unlike Bitcoin, can be used for multiple purposes [14], [12]. In Figure 1 we have an example of a Solidity contract, implementing the IERC20 interface.



```
contract ERC20 is IERC20 {
    using SafeMath for uint256;

    mapping(address => uint256) private _balances;

    mapping(address => mapping(address => uint256)) private _allowed;

    uint256 private _totalSupply;

    event Transfer(address indexed from, address indexed to, uint256 value);

    event Approval(
        address indexed owner,
        address indexed spender,
        uint256 value
    );

    function totalSupply() public view returns (uint256 supply) {
        return _totalSupply;
    }

    function balanceOf(address owner) public view returns (uint256 balance) {
        return _balances[owner];
    }

    function allowance(
        address owner,
        address spender
    ) public view returns (uint256 remaining) {
        return _allowed[owner][spender];
    }
```

Figure 1. Solidity contract example.

With the sharp growth of decentralized finance (DeFi), the average number of smart contracts deployed each month exceeded 4200 [11] from July 2020 to April 2021. According to DefiLlama statistics [7], the total amount blocked over DeFi protocols reached $230.8 billion on January 10, 2022.

A smart contract is a set of digital agreements and protocols within which the parties carry out their work [10]. The term “smart contract” was first proposed by Nick Szabo in 1994, who referred to smart contract as a computer system that enforces the conditions of a contract. It is a general purpose computational engine provided through the Ethereum Virtual Machine (EVM) and uses Ethereum blockchain concepts [10,5].

However, smart contracts are often prone to errors with potentially devastating financial effects [16]. The DAO bug [13] is an illustrative example of the difficulties involved in implementing a safe smart contract. A series of attacks is constantly launched to obstruct the natural flow or even completely destroy the network [2]. Attacks related to cryptocurrency wallets, smart contracts, transaction authentication, mining pools and blockchain networks are often exploited by adversaries. DAO, King of the Ether Throne, and Multiplayer attacks are some smart contract-based attacks that occur due to the bugs in the smart contract code [1].

Once deployed, ideally a smart contract is expected to be immutable, as it encodes an established contract and, as such, must not be modified. Implementation immutability, however, has two major drawbacks. First, contracts cannot be corrected if the implementation is found to be incorrect after being deployed. There are many examples of real-world contract instances failures that have been exploited with staggering sums of cryptocurrencies being taken over [15,4].

The DAO was a relatively small contract (2KLOC of Solidity code) that was heavily scrutinized by the wider Ethereum community prior to deployment. However, an attacker managed to exploit a subtle indentation bug to steal $60 million in cryptocurrency. Examples like the DAO highlight the mission critical nature of smart contracts. Although contract code is generally small by modern software standards, if the contract attracts a large amount of investment, the code carries a significant amount of value per line of code. Furthermore, since the contract code is stored on the blockchain, once deployed, the code is immutable and making updates or bug fixes is impossible without complex solutions involving a central authority [6].

Some studies support that a reliable deployer can be part of a process of deploying reactive systems in general, such as component-based, microservice-based systems or even systems of systems. This structural approach changes the immutability of a contract's implementation to its specification, promoting the "code is law" to the "specification is law" paradigm. It was evidenced that this paradigm shift brings a series of improvements [9].

Nevertheless, these specifications are still rarely used by developers due to the complexity of the notations and the process of formal conformance verification. Hence our work provides a bridge between a complex formal notation and natural language descriptions. We created a grammar to describe postconditions, more specifically post-conditions of ERC20 tokens, in order to help developers to turn descriptions of natural language functions into formal specifications. It is understood that scientific production in this sense is still scarce, despite the urgency and relevance of the matter.

In the following section, we introduce the relevant background material. Section 3 introduces the proposed grammar for translating formal postconditions into natural language descriptions, and vice versa, being done bidirectionally, from natural language to formal specification, while Section 4 presents the custom tool and the case study. Finally, Section 5 presents our conclusions, summarizes the work and discusses next steps.

2. Background

In order to appreciate the significance of the proposed grammar-based approach, it is essential to understand the role of formal methods in the development of complex computer systems, particularly in the realm of smart contracts. Formal methods, such as formal specification and verification, provide a rigorous and mathematically grounded way to describe, analyze, and verify computer systems, including smart contracts, ensuring their correctness and reliability. However, natural language processing techniques based on machine learning, which have made significant progress in understanding and generating human language, may introduce inaccuracies and uncertainties due to their probabilistic nature. This section delves into the details of formal methods and explores the limitations of machine learning-based natural language processing techniques, highlighting the need for a more deterministic approach like the grammar-based method proposed in this paper.

2.1. Formal methods

Formal methods refer to a set of techniques, tools, and methodologies used to describe, analyze, and verify complex computer systems by employing mathematical logic and formalized languages [28]. These methods aim to improve the reliability, safety, and security of systems by rigorously analyzing and verifying their properties and behavior. One can break down formal methods into specification and verification.

Formal specification involves expressing the requirements, design, or behavior of a system in a precise and unambiguous language based on mathematical logic. Such languages provide a formal syntax and semantics, enabling rigorous analysis and reasoning about the system's properties. Formal specifications can be used to describe functional and nonfunctional requirements, such as safety, security, or performance properties [29].

Formal verification is the process of proving the correctness of a system or validating its properties against a formal specification using mathematical techniques. This can be accomplished through model checking, theorem proving, or other formal analysis techniques. Formal verification ensures the absence of errors, vulnerabilities, or inconsistencies in the implementation, thus increasing the reliability and trustworthiness of the system [30].

In the context of this work, formal methods play a crucial role in ensuring the correctness and reliability of smart contracts. Since smart contracts are decentralized and self-executing, they often involve the transfer of significant financial assets and therefore must be error-free to avoid costly mistakes or exploitation by malicious actors [32].

In order to achieve a formal and deterministic semantics for smart contract specifications described in natural language, it is necessary to consider the limitations of natural language processing (NLP) techniques based on machine learning (ML). While ML-based NLP methods have made significant progress in understanding and generating human language, they are inherently probabilistic and rely on approximations [33]. This introduces uncertainty and potential inaccuracies that could lead to undesirable consequences in the context of formal specifications for smart contracts.

In contrast, grammar-based approaches offer a more deterministic way to process natural language descriptions and translate them into formal specifications. By leveraging well-defined rules and structures, grammar-based methods can provide a more precise and unambiguous translation between natural language and formal specifications, ensuring that the resulting smart contracts conform to their intended behavior and requirements.

2.2. Grammatical Framework

The Grammatical Framework (GF) is a distinctive grammar formalism that defines grammars through a combination of abstract and concrete syntax components [23]; it was the language used to create the proposed grammar. Based on Martin-Löf's type theory [24], the abstract syntax serves as a blueprint for constructing abstract syntax trees. On the other hand, the concrete syntax outlines linearization rules, which determine how these trees are transformed into expressions in a specific language. This emphasis on linearization rather than parsing sets GF apart from other grammar formalisms.

By employing multiple concrete syntaxes corresponding to the same abstract syntax, GF allows the representation of the same tree in different languages and facilitates translation within the defined language fragment [23]. The GF system [25] provides essential functions such as parsing and linearization, along with a syntax editor. This editor lets users interactively generate texts in various languages by manipulating abstract syntax trees and observing the results in a familiar language [26].

An integral part of the GF project is the GF Resource Grammar Library [27], which delivers an API for widely used linguistic structures. The library contains resource grammars for numerous languages, most of which share a similar interface. The resource grammar library streamlines the division of labor between domain specialists and linguistic experts. Domain specialists, who may not possess extensive linguistic knowledge, can create abstract syntax models for specific areas, connecting them to concrete languages using the resource grammars. Meanwhile, linguistic experts concentrate on implementing the resource grammars without requiring knowledge of any particular domain. This division of labor allows for effective collaboration in grammar engineering and fosters the development of precise language models within specialized domains.

Furthermore, grammar-based approaches can be extended with domain-specific rules and vocabulary to handle the unique characteristics and terminology of smart contracts and decentralized finance. This allows for a more accurate and tailored translation process, minimizing the risks of misinterpretation or ambiguity that could arise with ML-based NLP methods.

The proposed grammar-based approach to generating formal specifications from natural language descriptions aims to bridge the gap between human-readable documentation and machine-verifiable code. By transforming natural language function descriptions into formal specifications (and vice versa), developers can automatically derive postconditions from comments to verify conformance, or generate textual descriptions for complex formal annotations in existing smart contracts to improve legibility. This bi-directional translation support is a distinctive feature provided by GF.

2.3. Solc-Verify

Solc-verify is a software verification tool specifically designed to analyze Ethereum smart contracts written in the Solidity programming language. The primary motivation behind solc-verify is to proactively identify potential security vulnerabilities, correctness issues, and other discrepancies in smart contracts before they are deployed to the Ethereum blockchain. This is of critical importance since smart contracts are self-executing and (potentially) immutable, meaning that any errors or vulnerabilities present in the code can lead to severe financial and operational consequences once deployed [20]. Solc-verify leverages the Boogie intermediate verification language

(IVL) as a powerful and expressive means for representing and reasoning about Solidity contracts [19]. By converting Solidity code into Boogie, solc-verify can leverage existing verification techniques and tools, such as static checking, theorem proving, and symbolic execution, to systematically and rigorously verify the correctness and security of smart contracts.

The core component of solc-verify is the translation process, which converts Solidity source code into Boogie IVL. This process involves accurately representing various elements of Solidity contracts, such as data structures, functions, statements, and expressions in Boogie, which provides a suitable intermediate representation for performing verification tasks. One of the primary challenges in this translation process is to preserve the semantics of the original Solidity code while mapping it to the Boogie language.

Solc-verify is a tool that supports basic Solidity types and offers different modes for arithmetic operations, allowing users to choose the most suitable one for their needs. The simplest mode treats integers as unbounded mathematical integers, while more precise modes are available, such as SMT bitvectors and modular arithmetic. The tool also handles Solidity mappings and arrays using SMT arrays.

In solc-verify, Solidity functions are translated into Boogie procedures, with restrictions on state changes enforced by the compiler. Additionally, user-defined function modifiers can be applied to extend or modify the behavior of functions. The tool is capable of mapping most Solidity statements and expressions directly to their Boogie counterparts. Some transformations may be required, such as converting for loops to while loops or breaking down nested calls and assignments into separate statements.

Solc-verify also deals with Ethereum-specific transactions and balances, error handling, and overflow detection. The tool allows users to define high-level properties like contract invariants, loop invariants, pre- and post-conditions, and assertions, as annotations in solidity. It then leverages SMT solvers to verify contract properties in a modular, scalable, and user-friendly manner.

This approach provides precise and automated formal verification for Solidity smart contracts. Solc-verify is already applicable to real-world contracts, effectively identifying bugs and proving non-trivial properties with minimal user input [17].

2.4. Verification Framework

In [4], the limitations of smart contracts, particularly focusing on the "code is law" paradigm, are addressed, and a novel systematic deployment framework for enhancing their reliability and adaptability is proposed and implemented. The limitations are related to the immutability nature of smart contracts, as already discussed in the introduction.

To circumvent these limitations, the Ethereum community has adopted the proxy pattern, which simulates contract upgrades. Nevertheless, this method presents potential issues, such as not addressing the core problem of correctness and bestowing excessive power to contract maintainers. The authors of the framework presented in [4] propose a systematic deployment approach that mandates formal verification of smart contracts prior to their creation and upgrades. This framework, tailored for the Ethereum platform and smart contracts written in Solidity, is grounded in the design-by-contract methodology. It employs a trusted deployer to guarantee safe contract creations and updates, ensuring that the implementation conforms to the expected specification. As an off-chain service, this framework can seamlessly integrate into existing blockchain platforms, enabling participants to confirm the anticipated behavior of a contract.

As already mentioned, the proposed framework instigates a paradigm shift from "code is law" to "specification is law," reinforced through formal verification. This new paradigm effectively addresses the concerns of arbitrary code updates and prevents the deployment of defective contracts. It permits contracts to be optimized and adapted according to evolving business needs while preserving the assurance that implementations consistently conform to their corresponding specifications.

A prototype of this framework has been developed, and a case study involving real-world smart contracts derived from the ERC20, ERC3156, and ERC1155 Ethereum token standards was conducted. The results demonstrate promising performance, indicating that the new framework improves upon the existing "code is law" paradigm by concentrating on the more stable and crucial aspect of contract specifications, thereby better aligning with the standards of modern software engineering practices.

The framework focus lies in contract upgrades that preserve the signature of public functions, and it assumes that contract specifications fix the data structures used in the contract implementation. However, the authors of the framework intend to relax these restrictions in future versions. The framework's current emphasis is on partial correctness (loops are not addressed, and so termination of programs is not guaranteed), aligning with the goal of ensuring safety properties and the nature of smart contracts with explicitly bound executions. Extending the framework to address termination (total correctness) is also in the authors' agenda.

The authors of the framework suggest a specification format that delineates the required member variables and function signatures, accompanied by postconditions for function signatures and invariants for the specification. Unlike ordinary programs, public functions of smart contracts can be invoked by any well-formatted transaction. As a result, the proposed framework moves away from preconditions in the specification and requires postconditions to be met whenever public functions successfully terminate.

The framework currently requires that the postconditions are provided by the user.

3. Grammar

The grammatical framework grammar presented is designed to generate formal specifications for smart contracts, specifically those based on the ERC20 token standard, from textual descriptions in natural language. The grammar comprises several categories (cat) and functions (fun), which allow for the creation and manipulation of various expressions.

3.1. Abstract Grammar

The BNF table below provides a clear and concise overview of the grammar, outlining the production rules that define the structure of valid expressions in the language. The left column of the table lists the non-terminal symbols, which are italicized and enclosed in angle brackets (e.g., <Expression>, <Variable>, <BoolOperator>). These symbols represent syntactic categories and can be further decomposed using the production rules. The right column of the table contains the production rules, with terminal symbols shown in upper case (e.g., TotalSupply, Equals, Plus). Terminal symbols are the basic elements of the language and cannot be further divided.

```
<Description> ::= <Expression>

<Expression> ::= <Variable> <BoolOperator> <Variable>

            |<Expression>  <BoolOperator>  <Expression>  <BoolOperator>
<Variable>

            |<Expression> <Expression> <BoolOperator> <variable>

            |<Expression> <ArithmeticOperator> <Variable>

            |<Expression> <ArithmeticOperator> <Variable> <Expression>

            |<Variable> <BoolOperator> <vaVariableriable>

            |<BoolOperator> <Variable> <BoolOperator> <Variable>

<Variable> ::= totalSupply | supply | balancesOwner | balance | allowed
| remaining | senderAdress | senderAdressFrom | senderNewBalance |
senderNewBalanceFrom | recipientNewBalance | oldBalance | oldBalanceFrom
| oldBalanceRecipient | transferredValue | recipientAddress |
isSuccessful | isNotSuccessful | spenderAllowance | spenderAllowanceFrom
| spenderAllowanceFromMsgSender | spenderAllowanceAddressFrom |
specifiedValue | previousValue | previousValueAllowed

<BoolOperator> ::= equals | isNotEqual | and | or | lessThan |
greaterThan | lessThanOrEqual | greaterThanOrEqual

<ArithmeticOperator> ::= plus | minus
```

In the table below we can identify a mapping of the production rules that define the structure of valid expressions in the language.

Table 1. Mapping between abstract and concrete grammars

Abstract	ERC20Eng	ERC20Formal
Variable -> BoolOperator -> Variable -> Expression	the resulting total supply of tokens should be equal to the total supply of tokens	supply == _totalSupply
BoolOperator -> Variable -> BoolOperator -> Variable -> Expression	if the sender address (msg.sender) is equal to the recipient address	&& msg.sender == to
Expression -> BoolOperator -> Expression -> BoolOperator -> Variable -> Expression	the sender new balance (msg.sender) should be equal to their old balance (msg.sender) minus the transferred value if the sender address (msg.sender) is not equal to the recipient address or the sender new balance (msg.sender) should be equal to their old balance (msg.sender) if the sender address (msg.sender) is equal to the recipient address and the transfer is successful or the transfer is not successful	((_balances[msg.sender] == __verifier_old_uint(_balances[msg.sender])) - value) && msg.sender != to || (_balances[msg.sender] == __verifier_old_uint(_balances[msg.sender])) && msg.sender == to && success || !success

1. Categories:
    a. Description: Represents the top-level category for the generated formal specifications.
    b. Expression: Represents various expressions that can be formed using variables, operators, and other expressions.
    c. Variable: Represents the variables involved in the expressions, such as token balances and allowances.
    d. BoolOperator: Represents boolean operators like equal, not equal, and, or, etc.
    e. ArithmeticOperator: Represents arithmetic operators like addition and subtraction.
2. Functions:

The grammar contains several functions that create and manipulate expressions, enabling the formation of complex formal specifications.

3. Variables:

The grammar includes several variables representing elements of the ERC20 token standard, such as token balances, allowances, and addresses. Some examples are *totalSupply*, *balancesOwner*, *allowed*, *senderAdress*, and *transferredValue*.

4. Arithmetic and Boolean Operators:

The grammar supports basic arithmetic operators, like addition (plus) and subtraction (minus), and boolean operators like *equals*, *isNotEqual*, *and*, *or*, *lessThan*, *greaterThan*, *lessThanOrEqual*, and *greaterThanOrEqual*.

3.2. ERC20 Formal Concrete Grammar

This concrete grammar is designed to work with the abstract grammar previously discussed to generate formal specifications for ERC20 token-based smart contracts in a readable and human-understandable format. The concrete grammar comprises several linearizations (lin) of the categories and functions defined in the abstract grammar.

1. Linearization: Description, Expression, Variable, BoolOperator, and ArithmeticOperator: All of these categories are linearized as Str (String) types. This is because the formal specifications generated by this grammar are essentially strings representing different aspects of the smart contract's behavior.
2. Functions: The concrete grammar provides linearizations for each function in the abstract grammar. These linearizations essentially define how the different components of the formal specification will be combined to form a complete, human-readable representation. Some examples of these linearizations are:
    a. mkTransferFromSpenderExpression: This linearization concatenates the strings of exp1, boolOp, and exp2 to create a new Expression.
    b. mkApproveExpression: This linearization combines the strings of exp1, boolOp1, var1, boolOp2, exp2, boolOp3, and var2 to create a new Expression.
    c. mkComparativeExpression: This linearization concatenates the strings of var1, boolOp, and var2 to create a new Expression.

3.3. ERC20Eng Concrete Grammar

This grammar is an alternative concrete grammar to the previously provided formal specification concrete grammar, which is designed to generate more natural and human-readable formal specifications for ERC20 token-based smart contracts. The formal specification grammar employs the same abstract grammar but uses more natural language phrases and variations, making the generated specifications more understandable for non-experts.

The categories and linearizations in the formal specifications grammar are similar to the English concrete grammar, with a few differences:

1. Variable: In the formal specifications grammar, the Variable category is linearized as an object with two properties, 'name' and 'desc'. The 'name' property represents the actual variable used in the smart contract, while the 'desc' property provides a human-readable description of the variable.
2. BoolOperator and ArithmeticOperator: This English grammar uses the 'variants' function to provide multiple natural language alternatives for each operator. For example, “equals” can be represented as "should be equal to", "must be equal to", "is equal to", "should be the same as", or "must be the same as".
3. Functions: Similar to the Formal concrete grammar, the formal specifications grammar provides linearizations for each function in the abstract grammar. These linearizations define how the different components of the formal specification will be combined to form a complete, human-readable representation. However, in this grammar, the linearizations use more natural language phrases and variations.

4. Tool support and case study

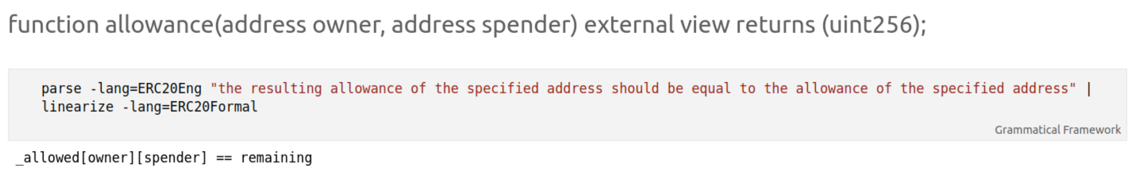
This section presents the tool support provided by our approach and demonstrates its functionality through a case study. Our system facilitates the creation of postconditions for ERC20 token functions in smart contracts, enabling developers to use these notations for testing purposes within the solc-verify framework [39]. The grammar and functionality are illustrated in a Jupyter notebook [37], which is hosted on Binder [36], a service that allows for the seamless creation and sharing of Jupyter Notebooks without the need for local software installation.

4.1. Jupyter Notebook and Binder

Jupyter Notebooks are interactive, web-based computational environments widely used in data science, machine learning, and scientific research. They enable users to create and share documents containing live code, equations, visualizations, and narrative text for various purposes, such as data cleaning, transformation, visualization, and modeling [37].

Binder, powered by the open-source BinderHub project, provides the ability to deploy Jupyter Notebooks in the cloud using container technology. It allows users to specify a GitHub repository containing the Jupyter Notebook and required dependencies, creating a Docker [39] container (an isolated environment) with the specified configuration, and launching the Jupyter Notebook within it. Users can access the container through a web browser and interact with the notebook without installing any software on their own computers.

In Figure 1, we have an example of how the Binder integrated with the Jupyter Notebook (GF-binder) works, where we perform a linearization command from the Grammatical Framework, with the aim of generating a postcondition for the allowance function of the ERC20 token.


function allowance(address owner, address spender) external view returns (uint256);

```
parse -lang=ERC20Eng "the resulting allowance of the specified address should be equal to the allowance of the specified address" |
linearize -lang=ERC20Formal
```

```
_allowed[owner][spender] == remaining
```

Figure 2. linearization command in GF-binder to generate the postcondition of the ERC20 allowance function.

4.2. Case Study

The case study demonstrates the supported functions, parameters, return values, and accepted phrases and words in the grammar designed to aid smart contract developers in creating postconditions for ERC20 token functions. The solc-verify framework [39] utilizes these postcondition notations for testing purposes.

The Jupyter notebook, which hosts the grammar, can be found in a Github repository [40]. It provides an interactive environment for developers to explore the functionality of the grammar, generate formal specifications from natural language descriptions, being possible to be done in a bidirectional way, from natural language to formal specification, and vice versa.

4.2.1. Supported Functions

The following functions are supported within our system, with their respective function signatures and postconditions for both the natural language (ERC20Eng) and formal language (ERC20Formal) grammars:

1. totalSupply
    - Function signature: function totalSupply() external view returns (uint256);
    - Postcondition:
        - ERC20Eng: "the resulting total supply of tokens should be equal to the total supply of tokens".
        - ERC20Formal: "supply == _totalSupply".
2. balanceOf
    - Function signature: function balanceOf(address who) external view returns (uint256);
    - Postcondition:

    - ERC20Eng: "the resulting balance of the specified address should be equal to the balance of the specified address".
    - ERC20Formal: "_balances[owner] == balance".

3. allowance

    - Function signature: function allowance(address owner, address spender) external view returns (uint256);
    - Postcondition:
        - ERC20Eng: "the resulting allowance of the specified address should be equal to the allowance of the specified address".
        - ERC20Formal: "_allowed[owner][spender] == remaining".

In the context of the Grammatical Framework (GF), a tree is a representation of the syntactic structure of a linguistic expression. A tree in GF consists of nodes and edges, where nodes represent linguistic categories (e.g., noun phrases, verb phrases, or sentences) and edges represent grammar rules that combine these categories. The root of the tree is the top-level linguistic category, typically a sentence, and the leaf nodes represent individual words or morphemes. A tree representing the allowance function structure can be seen in Figure 1.

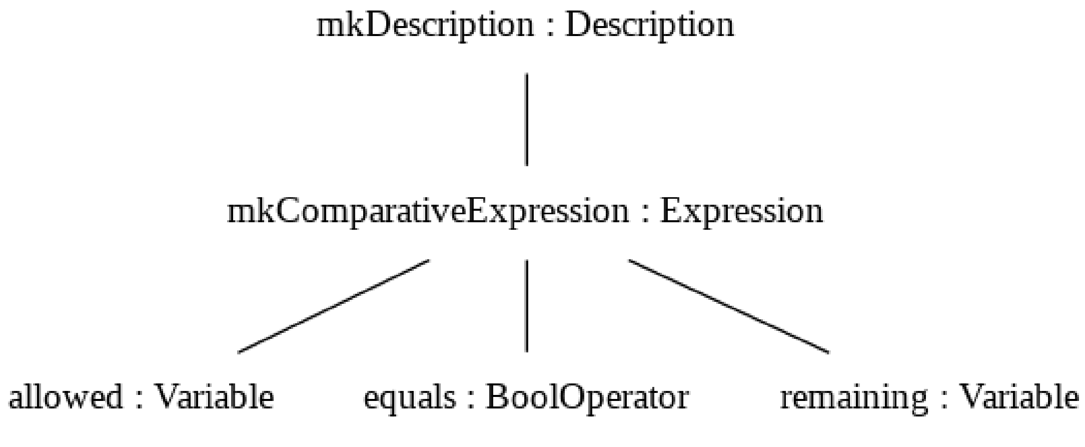

mkDescription : Description
mkComparativeExpression : Expression
allowed : Variable
equals : BoolOperator
remaining : Variable

Figure 3. tree syntactic structure of the allowance function.

On the other hand, the term "parse" refers to the process of analyzing a sentence to identify its syntactic structure. In GF, this process results in one or more possible syntactic trees, depending on the language and the ambiguity of the input sentence.

It provides a visual representation of the identified structure(s) and can be helpful for understanding the way a sentence has been parsed according to the grammar rules defined in the GF module. This visualization can be particularly useful for debugging and refining grammars or for educational purposes to illustrate the syntactic structure of different languages.

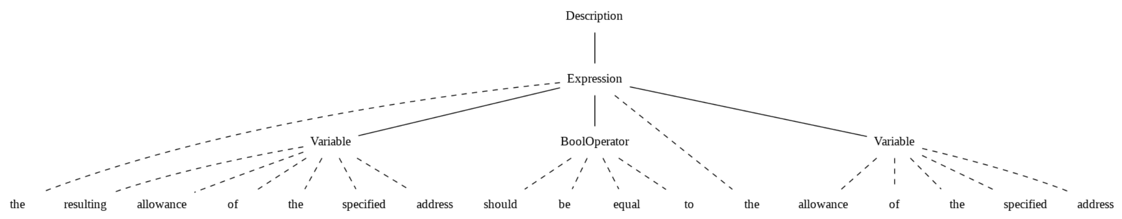


Figure 4. parse syntactic structure of the allowance function.

4. transfer

- Function signature: function transfer(address to, uint256 value) external returns (bool);
- Postconditions:
  - ERC20Eng: "the sender new balance (msg.sender) should be equal to their old balance (msg.sender) minus the transferred value if the sender address (msg.sender) is not equal to the recipient address or the sender new balance (msg.sender) should be equal to their old balance (msg.sender) if the sender address (msg.sender) is equal to the recipient address and the transfer is successful or the transfer is not successful".
  - ERC20Formal: "((_balances[msg.sender] == __verifier_old_uint(_balances[msg.sender])) - value) && msg.sender != to || (_balances[msg.sender] == __verifier_old_uint(_balances[msg.sender])) && msg.sender == to && success || !success".
  - ERC20Eng: "the recipient new balance should be equal to their old balance (msg.sender) plus the transferred value if the sender address (msg.sender) is not equal to the recipient address or the recipient new balance should be equal to their old balance (msg.sender) if the sender address (msg.sender) is equal to the recipient address and the transfer is successful or the transfer is not successful".
  - ERC20Formal: "((_balances[to] == __verifier_old_uint(_balances[msg.sender])) + value) && msg.sender != to || (_balances[to] ==

__verifier_old_uint(_balances[msg.sender])) && msg.sender == to && success || !success”.

5. approve

   - Function signature: function approve(address spender, uint256 value) external returns (bool);

   - Postcondition:

     - ERC20Eng: "the spender allowance (msg.sender)(spender) should be equal to the specified value and the operation is successful or the spender allowance (msg.sender)(spender) should be equal to the previous value if the operation is not successful”.
     - -ERC20Formal: "(_allowed[msg.sender][spender] == value && success) || (_allowed[msg.sender][spender] == __verifier_old_uint (_allowed[msg.sender][spender]) && !success)".

6. transferFrom

   - Function signature: function transferFrom(address from, address to, uint256 value) external returns (bool);

   - Postconditions:

     - ERC20Eng: "the sender new balance (from) should be equal to their old balance (from) minus the transferred value if the sender address (from) is not equal to the recipient address or the sender new balance (from) should be equal to their old balance (from) if the sender address (from) is equal to the recipient address and the transfer is successful or the transfer is not successful".

     - ERC20Formal: "((_balances[from] == __verifier_old_uint(_balances[from])) - value) && from != to ||

(_balances[from] == __verifier_old_uint(_balances[from])) && from == to && success || !success”.

- ERC20Eng: "the recipient new balance should be equal to their old balance (to) plus the transferred value if the sender address (from) is not equal to the recipient address or the recipient new balance should be equal to their old balance (from) if the sender address (from) is equal to the recipient address and the transfer is successful or the transfer is not successful".
- ERC20Formal: "((_balances[to] == __verifier_old_uint(_balances[to])) + value) && from != to || (_balances[to] == __verifier_old_uint(_balances[from])) && from == to && success || !success”.
- ERC20Eng: "the spender allowance (from)(msg.sender) should be equal to the previous value (from)(msg.sender) minus the transferred value and the operation is successful or the spender allowance (from)(msg.sender) should be equal to the previous value (from)(msg.sender) and the operation is not successful or the spender allowance address (from) is equal to the sender address (msg.sender)".
- ERC20Formal: "((_allowed[from][msg.sender] == __verifier_old_uint(_allowed[from][msg.sender]) - value) && success) || (_allowed[from][msg.sender] == __verifier_old_uint(_allowed[from][msg.sender])) && !success) || from == msg.sender”.
- ERC20Eng: "the spender allowance (from)(msg.sender) should be less than or equal to the previous value (from)(msg.sender) or the spender allowance (from) is not equal to the sender address (msg.sender)".

- ERC20Formal: "_allowed[from][msg.sender] <= __verifier_old_uint(_allowed[from][msg.sender]) || from != msg.sender".

4.2.2. Supported Tokens

Here is the catalog of possible articles, verbs, boolean operators, variables, and arithmetic operators, in the English language:

Table 2. Grammar supported tokens.

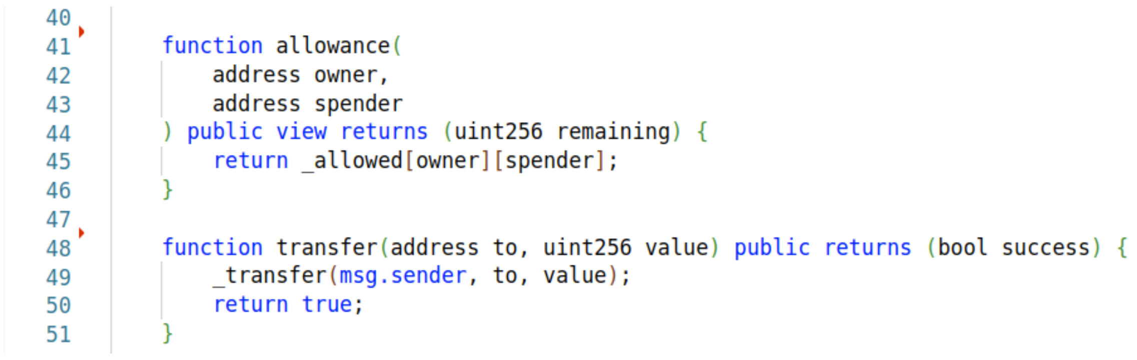
	Tokens
Articles	"the", "their".
Verbs	"should be", "must be", "is".
Boolean operators	Equals: "should be equal to", "must be equal to", "is equal to", "should be the same as", "must be the same as"; Not Equals: "is not equal to", "should be different from", "must be different from", "must not be equal to", "should not be equal to", "should not be the same as", "must not be the same as"; Less Than: "is less than", "should be less than", "must be less than"; Greater Than: "is greater than", "should be greater than", "must be greater than"; Less Than Or Equals: "is less than or equal to", "should be less than or equal to", "must be less than or equal to"; Greater Than Or Equals: "is greater than or equal to", "should be greater than or equal to", "must be greater than or equal to"; Others: "and" "if" "or".
Arithmetic operators	"plus", "minus".
Variables	"resulting total supply of tokens", "total supply of tokens", "resulting balance of the specified address", "balance of the specified address", "resulting allowance of the specified address", "allowance of the specified address", "sender address (msg.sender)", "sender address (from)" "recipient address", "sender new balance

	(msg.sender)", "sender new balance (from)", "recipient new balance", "old balance (msg.sender)", "old balance (from)", "old balance (to)", "transferred value", "spender allowance (msg.sender)(spender)", "spender allowance (from)", "spender allowance (from)(msg.sender)", "spender allowance address (from)", "specified value", "previous value", "previous value (from)(msg.sender)".

4.2.3. Annotated contracts

Here we have some examples of what contracts written in Solidity looked like before and after the adoption of post-condition descriptions of ERC20 functions.

Contracts previously annotated with postconditions by the authors of the framework [4] mentioned in previous sections were used. These contracts can be found in [41]. In Figure 3 we have the ERC20 contract that extends the IERC20 interface.

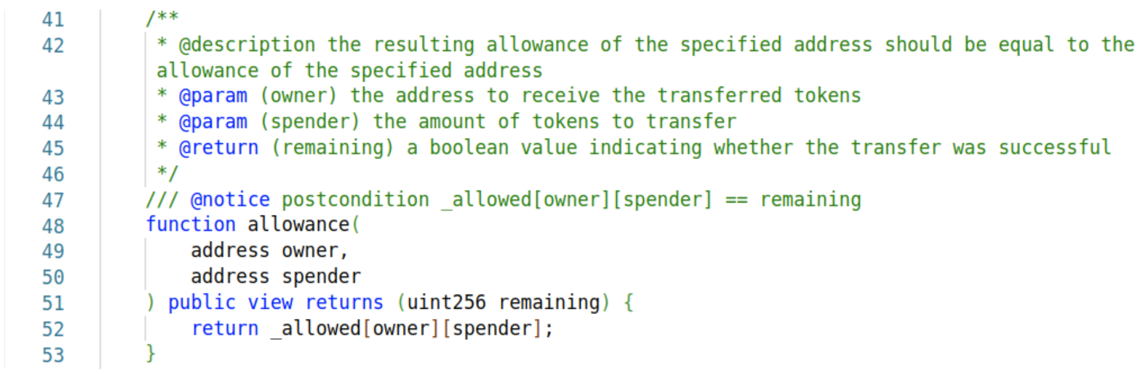


```

    function allowance(
        address owner,
        address spender
    ) public view returns (uint256 remaining) {
        return _allowed[owner][spender];
    }

    function transfer(address to, uint256 value) public returns (bool success) {
        _transfer(msg.sender, to, value);
        return true;
    }
```

Figure 5. allowance and transfer functions without postcondition annotations.

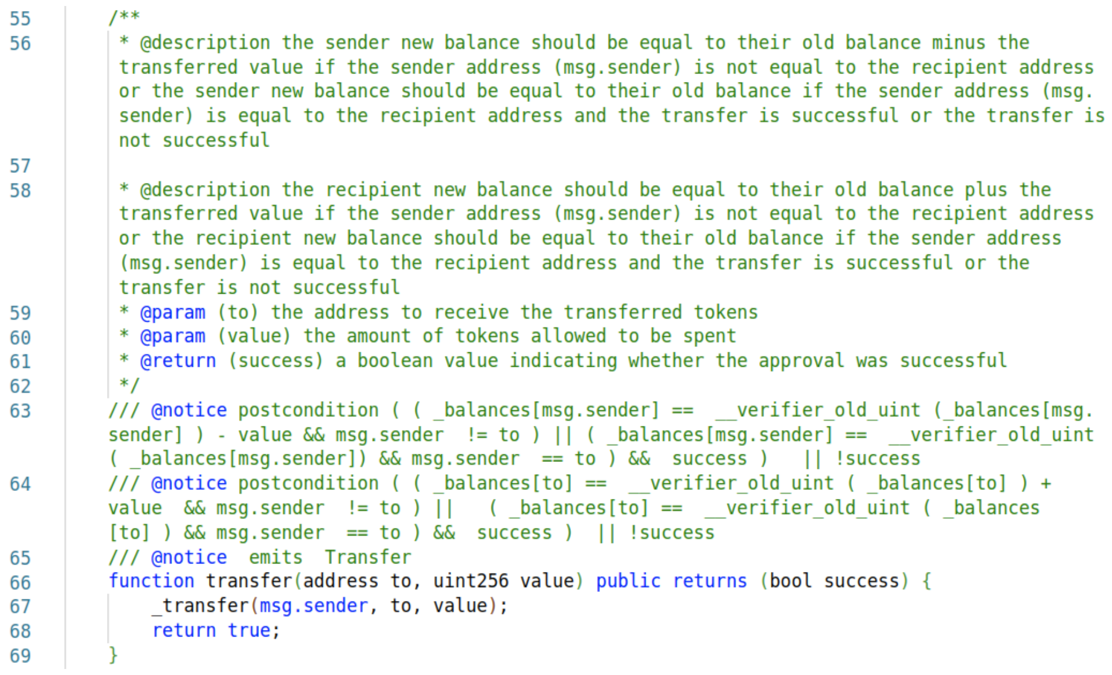
By convention and ease of identification, the notation "@description" was created to identify a description in natural language made in the English language. To identify postconditions, the authors of the framework used the notation "@notice postcondition" to express a postcondition notation.

In Figures 4 and 5 we have the contracts annotated with the descriptions in English natural language, initially manually inferred by us, which consequently generated the grammars. The "@param" notations are not used in the grammar at the moment, but we intend to use them to bring more flexibility to the tool in future work.



```
/**
 * @description the resulting allowance of the specified address should be equal to the
 allowance of the specified address
 * @param (owner) the address to receive the transferred tokens
 * @param (spender) the amount of tokens to transfer
 * @return (remaining) a boolean value indicating whether the transfer was successful
 */
/// @notice postcondition _allowed[owner][spender] == remaining
function allowance(
    address owner,
    address spender
) public view returns (uint256 remaining) {
    return _allowed[owner][spender];
}
```

Figure 6. allowance function with postcondition descriptions and formal specification annotations.



```
/**
 * @description the sender new balance should be equal to their old balance minus the
 transferred value if the sender address (msg.sender) is not equal to the recipient address
 or the sender new balance should be equal to their old balance if the sender address (msg.
 sender) is equal to the recipient address and the transfer is successful or the transfer is
 not successful

 * @description the recipient new balance should be equal to their old balance plus the
 transferred value if the sender address (msg.sender) is not equal to the recipient address
 or the recipient new balance should be equal to their old balance if the sender address
 (msg.sender) is equal to the recipient address and the transfer is successful or the
 transfer is not successful
 * @param (to) the address to receive the transferred tokens
 * @param (value) the amount of tokens allowed to be spent
 * @return (success) a boolean value indicating whether the approval was successful
 */
/// @notice postcondition ( ( _balances[msg.sender] ==  __verifier_old_uint (_balances[msg.
sender] ) - value && msg.sender  != to ) || ( _balances[msg.sender] ==  __verifier_old_uint
( _balances[msg.sender]) && msg.sender  == to ) &&  success )   || !success
/// @notice postcondition ( ( _balances[to] ==  __verifier_old_uint ( _balances[to] ) +
value  && msg.sender  != to ) ||   ( _balances[to] ==  __verifier_old_uint ( _balances
[to] ) && msg.sender  == to ) &&  success )  || !success
/// @notice  emits  Transfer
function transfer(address to, uint256 value) public returns (bool success) {
    _transfer(msg.sender, to, value);
    return true;
}
```

Figure 6. transfer function with postcondition description and formal specification notations.

5. Related Work

In [34] the authors propose a technique for automatically generating formal program specifications from natural language comments inserted in code. To accomplish this, textual comments are preprocessed by first removing special characters, then inserting spaces between words, and finally converting all words to lowercase. The preprocessed comments are then split into individual words using space. The intermediate representation (IR) translator is then used to generalize the Java modeling language (JML) specification in a textual form to an abstract form and parse the abstracted specification to an Intermediate Representation (IR) using the Abstract Syntax Tree (AST). The use of IR helps to reduce the search space for synthesis and facilitate the instantiation of the synthesis results with concrete information belonging to the target method.

To assess the accuracy of the generated specifications, the authors evaluated their technique on 5 representative projects with diverse functionalities such as graph handling and efficient data structures. Each generated specification was manually checked against the source code by two developers, and their correctness was verified by running developer-written test cases. Their evaluation results demonstrate that the generated specifications precisely represent the method behaviors and improve the efficiency and effectiveness of various analysis and testing applications.

Their approach assumes good quality comments, and their empirical studies have shown that over 50% of comments/documentation are of good quality and useful in practice. Furthermore, their approach has the potential to improve the quality of and robustness of software systems which aligns with the goal of generating formal specifications for smart contracts to reduce significant financial consequences caused by errors in code.

Their evaluation results demonstrated the usefulness of their generated specifications in dynamic testing and analysis, which could be applicable to testing and analyzing smart contracts. Moreover, they "assembled primitive tokens guided by specification syntax and properties of the target method" to synthesize program specification candidates. This process could potentially inspire our proposed framework in generating smart contract specifications that consider a wider range of perspectives and exceptions.

Thus, their work on generating formal program specifications from natural language comments provides valuable insights that could potentially benefit our research on generating formal specifications for smart contracts from natural language textual descriptions.

Another related work [35] presents a system for automatically translating formal software specifications to natural language using the Grammatical Framework (GF), The system produces natural language that is acceptable to a human reader while allowing for optimization by users who are not experts in the system. The system is built around a GF grammar for specifications, consisting of an abstract syntax that provides rules for forming abstract syntax trees of specifications and three concrete syntaxes to present abstract syntax trees in OCL, English, and German.

The authors present a non-trivial case study that consists of specifications for the Java Card API translated into English by their system, which demonstrates their approach can scale up to handle complex real-world specifications.

While their work focuses specifically on translating formal software specifications to natural language text, their use of GF could provide valuable insights for our research on generating formal specifications for smart contracts from natural language textual descriptions. Additionally, their approach of formatting and automatic generation of grammar modules for domain-specific vocabulary could apply to generating formal specifications for smart contracts, where specialized domain-specific vocabulary is used.

Furthermore, their work on integrating formal software specification and verification into the industrial software engineering process aligns with the goal of generating formal specifications for smart contracts to ensure their quality and security.

6. Conclusion and Future Work

The proposed grammar for generating formal specifications of smart contracts from natural language textual descriptions aims to bridge the gap between natural language processing and formal verification in the domain of smart contracts. By leveraging grammars, this approach facilitates the generation of accurate, complete, and human-readable formal specifications for smart contracts, particularly those based on the ERC20 token standard.

The evaluation of the proposed grammar demonstrates its effectiveness in generating formal specifications that are correct, readable, and complete. Furthermore, the study of related approaches, such as techniques for generating formal program specifications from natural language comments in code and translating formal software specifications to natural language using the Grammatical Framework (GF), provides valuable insights that could potentially benefit the research on generating formal specifications for smart contracts from natural language textual descriptions.

Future work in this area could include extending the proposed grammatical framework to support additional smart contract standards and platforms, as well as exploring the integration of the proposed framework with existing formal verification tools and techniques to further enhance the safety and security of smart contracts. Additionally, future research could focus on improving the grammar's ability to handle more complex and diverse natural language inputs, as well as investigating the possibility of generating formal specifications in multiple languages.

We envision integrating the grammar into a tool that extracts information from Solidity smart contracts, reads the functions and descriptions crafted by developers, and automatically annotates the contract with post-conditions. This integration could substantially streamline the process of generating formal specifications from natural language descriptions, further enhancing the safety and reliability of smart contracts.

In conclusion, this work represents a promising approach to improve the quality and safety of smart contracts, as well as to facilitate their understanding and verification by both experts and non-experts alike.

References

1. N. Atzei, M. Bartoletti and T. Cimoli, "A survey of attacks on ethereum smart contracts", *Proc. Int. Conf. Princ. Secur. Trust*, pages 164-186, 2017.

2. A. Soundararajan, 10 Blockchain and New Age Security Attacks You Should Know, Jul. 2019, [online] Available: https://blogs.arubanetworks.com/solutions/10-blockchain-and-new-age-security-attacks-you-should-know/.

3. Solidity — solidity 0.8.20 documentation. [Online]. Available: https://media.readthedocs.org/pdf/solidity/develop/ solidity.pdf

4. Pedro Antonino, Juliandson Ferreira, Augusto Sampaio, and A. W. Roscoe. Specification is Law: Safe Creation and Upgrade of Ethereum Smart Contracts. In Software Engineering and Formal Methods: 20th International Conference, SEFM, Berlin, Germany, September 2022, Proceedings. Springer-Verlag, pages 227–243, 2022.

5. Ma F, Fu Y, Ren M, Wang M, Jiang Y, Zhang K, Li H, Shi X. EVM*: from offline detection to online reinforcement for ethereum virtual machine. In *IEEE 2019*, 26th international conference on software analysis, evolution and reengineering (SANER), pages 554-558, 2019.

6. David Siegel. Understanding the dao attack. https://www.coindesk.com/ understanding-dao-hack-journalists accessed on 22 July 2021.

7. DefiLlama, URL https://defillama.com/, (Acessado em 17 de abril de 2023).Google Scholar

8. M. Wohrer and U. Zdun. Smart contracts: security patterns in the ethereum ecosystem and solidity. International Workshop on Blockchain Oriented Soft. Engineering (IWBOSE) 2018, pages 2-8. Campobasso, 2018.

9. Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In POST 2017, pages 164–186. Springer, 2017.

10. Hoffmann T. Smart contracts and void declarations of intent. In: International conference on advanced information systems engineering, pages 168-175. Springer, 2019.

11. https://duneanalytics.com/,(Acessado em 17 de abril de 2023).Google Scholar

12. Alsayed Kassem J., Sayeed S., Marco-Gisbert H., Pervez Z., Dahal K., DNS-IdM: A Blockchain Identity Management System to Secure Personal Data Sharing in a Network. *Appl. Sci*,*9*,2953, 2019.

13. Dhillon, V., Metcalf, D., Hooper, M.: The DAO hacked. In: Dhillon, V., Metcalf, D., Hooper, M. (eds.) Blockchain Enabled Applications. pages 67–78. Apress, Berkeley, 2017.

14. UMA. Rosic, What is Ethereum?. outubro de 2016, [online] Disponível: https://blockgeeks.com/guides/ethereum/.

15. Jeannerod, N., Marché, C., Treinen, R.: A Formally Verified Interpreter for a Shell-like Programming Language. In: 9th Working Conference on Verified Software: Theories, Tools, and Experiments, volume 10712 of Lecture Notes in Computer Science, 2017.

16. Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In POST 2017, pages 164–186. Springer, 2017.

17. Ákos Hajdu and Dejan Jovanović. solc-verify: A modular verifier for solidity smart contracts. In VSTTE, pages 161–179. Springer, 2020.

18. Ákos Hajdu and Dejan Jovanović. Smt-friendly formalization of the solidity memory model. In ESOP 2020, pages 224–250. Springer, 2020.

19. Antonopoulos, A., Wood, G.: Mastering Ethereum: Building Smart Contracts and DApps. O'Reilly Media, Inc., Sebastopol, 2018.

20. DeLine, R., Leino, K.R.M.: BoogiePL: a typed procedural language for checking

object-oriented programs. Technical report MSR-TR-2005-70, Microsoft Research, 2005.

21. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M

Leino. Boogie: A modular reusable verifier for object-oriented programs. In FMCO 2005, pages 364–387. Springer, 2005.

22. Antonino P, Ferreira J, Sampaio A, W RA. Specification is Law: Safe Creation and Upgrade of Ethereum Smart Contracts [Internet]. arXiv.org. 2022 [cited 2023 Apr 24]. Available from: https://arxiv.org/abs/2205.07529

23. Ranta, A.: Grammatical Framework: A Type-theoretical Grammar Formalism.

The Journal of Functional Programming 14 (2004) 145–189

24. Martin-L¨of, P.: Intuitionistic Type Theory. Bibliopolis, Napoli (1984)

25. Ranta, A.: Grammatical Framework homepage (2005) www.cs.chalmers.se/ ~aarne/GF.

26. Khegai, J., Nordstr¨om, B., Ranta, A.: Multilingual syntax editing in GF. In

Gelbukh, A., ed.: CICLing-2003, Mexico City, Mexico. LNCS, Springer (2003)

27. Ranta, A.: The GF resource grammar library (2004) http://www.cs.chalmers.se/ ~aarne/GF/lib/resource/.

28. Woodcock J, Davies J. Using Z: Specification, Refinement, and Proof. Prentice Hall International; 1996.

29. Wing JM. A Specifier's Introduction to Formal Methods. Computer. 1990;23(9):8-24.

30. Baier C, Katoen JP. Principles of Model Checking. MIT Press; 2008.

31. Ambrus Á, Biczók G. Formal Specification and Verification of Solidity Contracts with Events. In: Proceedings of the 2nd ACM Conference on Advances in Financial Technologies. Zurich, Switzerland; 2020. p. 1-11.

33. Goldberg Y. A Primer on Neural Network Models for Natural Language Processing. Journal of Artificial Intelligence Research.

34. Zhai, Juan et al. “C2S: translating natural language comments to formal program specifications.” Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (2020)

35. David A. Burke and Kristofer Johannisson. 2005. Translating formal software specifications to natural language. In Proceedings of the 5th international conference on Logical Aspects of Computational Linguistics (LACL'05). Springer-Verlag, Berlin, Heidelberg, 51–66.

36. The Binder Project [Internet]. Mybinder.org. 2023 [cited 2023 Apr 28]. Available from: https://mybinder.org/

37. Jupyter Project Documentation — Jupyter Documentation 4.1.1 alpha documentation [Internet]. Jupyter.org. 2023 [cited 2023 Apr 28]. Available from: https://docs.jupyter.org/en/latest/

38. Docker Docs: How to build, share, and run applications [Internet]. Docker Documentation. 2023 [cited 2023 Apr 28]. Available from: https://docs.docker.com/

39. SRI-CSL. solidity/SOLC-VERIFY-README.md at 0.7 · SRI-CSL/solidity [Internet]. GitHub. 2023 [cited 2023 Apr 28]. Available from: https://github.com/SRI-CSL/solidity/blob/0.7/SOLC-VERIFY-README.md

40. gabrielnogueiralt. gabrielnogueiralt/gf-binder: Create and edit GF grammars in Jupyter notebook, hosted at https://mybinder.org/ [Internet]. GitHub. 2023 [cited 2023 Apr 28]. Available from: https://github.com/gabrielnogueiralt/gf-binder

41. gabrielnogueiralt. gabrielnogueiralt/safeevolutionrefinement [Internet]. GitHub. 2015 [cited 2023 Apr 28]. Available from: https://github.com/gabrielnogueiralt/safeevolutionrefinement