Extracting formal smart-contract specifications from natural language with LLMs

Gabriel Leite[1], Filipe Arruda[1][0009−0008−1111−9142], Pedro Antonino[2][0000−0002−5627−0910], Augusto Sampaio[1][0000−0001−6593−577X], and A .W. Roscoe[2,3][0000−0001−7557−3901]

[1] Centro de Informática, Universidade Federal de Pernambuco, Brazil
{gnl2,fmca,acas}@cin.ufpe.br
[2] The Blockhouse Technology Limited, UK
pedro@tbtl.com
[3] University College Oxford Blockchain Research Centre, UK
awroscoe@gmail.com

Abstract. Developers tend to be reluctant to provide formal specifications for software components; even well-established design-by-contract (DbC) properties like invariants, pre- and postconditions are neglected. This has hindered a more widely practical dissemination of the DbC paradigm. In this paper, we employ state-of-the-art NL processing technologies, using Large Language Models (LLMs), particularly, ChatGPT, to automatically infer formal specifications from component textual behavioural descriptions. More specifically, we implemented a framework (DbC-GPT), parameterised by a context, which is able to generate post-condition specifications for smart contract functions implemented in Solidity. The output of DbC-GPT is in the notation of the solc-verify tool (a verifier for Solidity) that is used to: (i) check the syntax of the inferred specification; and (ii) verify whether a reference implementation conforms to this specification. This is carried out in a loop in such a way that the DbC-GPT context is iteratively improved with verification counterexamples. To evaluate DbC-GPT, we have used some Ethereum standards (ERC20, ERC721, and ERC1155) and compared the precision of the generated specifications for several GPT contexts that consider information of these standards in isolation as well as their combination.

Keywords: smart contracts, natural language processing, design-by-contract, formal verification, LLMs, GPT, ChatGPT

1 Introduction

The development of smart contracts is marked by their immutable and autonomous nature, which demands high precision in their specification and implementation. While there are several approaches to specify and reason about smart contracts [26, 15, 12, 17, 5], the widespread integration of formal methods into software development processes continues to be a significant challenge.

An approach that has gained progressively more attention to tackle this issue is *hidden formal methods*, whose goal is to make formal methods more accessible and practical by integrating them seamlessly, and in a way as transparent as possible, into existing development workflows. The usual starting point of such approaches are requirements expressed in natural language (NL). By using NL processing (NLP) techniques, these approaches transform NL requirements into more precise models that can then be used for code generation [11, 10], formal verification [7, 9], testing [25, 16, 6], and so on.

Large Language models (LLMs), including GPT (Generative Pre-trained Transformer) variants, utilise deep learning to capture the nuances of natural language, making them particularly suited for tasks that require a deep semantic understanding of text, significantly reducing the risk of human error and misinterpretation [24]. Furthermore, these AI-driven approaches support continuous learning and adaptation, which allows them to improve over time as they are exposed to more examples and potential corrections [8]. Some works have recently used LLMs to automate the extraction and processing of text for several different purposes, including the generation of formal specifications from requirements documents [14]. Some other approaches have used LLMs to infer specification elements directly from implementations, such as the generation of loop invariants from C programs [13].

Despite the several efforts in this direction, the overall question of to what extent LLMs can systematically support formal specification and reasoning in software development needs to be more deeply investigated.

The main purpose of this paper is to integrate these transformer models (specifically ChatGPT [8]) into a framework, that we call DbC-GPT, which can automate the extraction of formal specifications for smart contracts from natural language descriptions. Although this can be generally applied to different languages, we focus on Solidity [1]. Specifically, DbC-GPT generates specifications in the design-by-contract paradigm, with focus on postconditions for each function of a Solidity smart contract. Preconditions and postconditions share the same nature as they both serve as formal specifications defining the expected properties or constraints at different points of function execution—before and after, respectively. We consider only postconditions because we reason about contracts as open programs since there is hardly any control concerning the caller of a contract deployed in a blockchain, and a precondition imposes a responsibility to be obeyed by the caller, which, in this context, is not generally possible to verify. We consider invariants as a topic for future work.

By focusing typically on coding, developers often hesitate to supply formal specifications, hindering a broader practical adoption of the design-by-contract paradigm. Furthermore, even when specifications are provided, an automated framework, as the one proposed here, helps to mitigate risks associated with manual encoding of these specifications from requirements, which tends to be error-prone. In the particular context of our research to build a fully automatic trusted deployer for smart contracts running in blockchains [3, 4], DbC-GPT fills an important gap since it automatically generates the specifications that are fed

into the trusted deployer to ensure a safe deploy and subsequent evolution of smart contract implementations.

The DbC-GPT framework takes three inputs: an Ethereum Improvement Proposal (EIP), an interface (with the functions to be specified), and a reference Solidity implementation for this standard. Its output is a design-by-contract specification in the notation of the solc-verify tool (a verifier for Solidity); as a constraint, the reference implementation provided as input must conform to the produced specification. Conformance is the usual (partial) correctness notion adopted in DbC approaches: assuming the precondition (in our context, this is always the true predicate) of a function holds, its execution must obey the respective postcondition, provided the execution terminates. The framework is parameterised by a (LLM) model and a demonstration context. In this paper, we use ChatGPT's GPT-4o model and contexts, but we could have instantiated our framework with Llama [23] instead, for instance.

For evaluation purposes, we have explored 8 demonstration contexts, which are differentiated in the way we provide information about the Ethereum standards. These examples contain example reference specifications of these standards so that the ChatGPT model has instances of what type of specification it should generate. We use solc-verify to check for conformance of the reference implementation with respect to the generated specification. This verifier is also used to check the syntax of the inferred specification. For all these contexts, verification is used in a feedback loop so that counterexamples (or verification errors, in general) are used to extend the respective learning context. We then compared these instances of the framework, each instantiated with one of the GPT contexts, concerning the precision of the generated specifications.

A distinguishing feature of our approach is the verification loop in the process. What we propose in this paper is a sort of counterexample-guided specification generation framework. We show that this provides relevant feedback information to improve the context so that progressively more precise specifications are generated by DbC-GPT. This design is based on the idea of *in-context learning* [8], namely, that GPT models perform well when the learning context is extended to refine the scope of the task being solved.

The next section provides some background on Solidity and solc-verify based on a small fragment of the Ethereum ERC20 standard that we also use as a running example. Section 3 provides an overview of the proposed framework and details of its design as a generative specification mechanism parameterised by customised GPT (demonstration) contexts. The evaluation of the framework with some Ethereum standards is presented in Section 4. Related work is addressed in Section 5. We then conclude with a summary of the contributions, limitations, and topics for future work.

2 Background

The ERC20 is a standard interface for fungible tokens on the Ethereum blockchain. It includes several key functions, such as transferring tokens, approving al-

transfer

Transfers `_value` amount of tokens to address `_to`, and MUST fire the `Transfer` event. The function SHOULD `throw` if the message caller's account balance does not have enough tokens to spend.

Note Transfers of 0 values MUST be treated as normal transfers and fire the `Transfer` event.

```
function transfer(address _to, uint256 _value) public returns (bool success)
```

Fig. 1. EIP informal specification for the `transfer` function

lowances, and transferring tokens on behalf of others. The description of an Ethereum standard is given by a document called Ethereum Improvement Proposals (EIP) such as, for example, that for the ERC20[4]. This includes the standard interface for each function, and an associated informal specification. An example is given for the ERC20 function `transfer` in Figure 1. This function has two input parameters (`_to` for the receiver address and `_value` for the amount to be transferred); as explained in the sequel, the address for the sender is an implicit parameter in Solidity. There is also a boolean output parameter (`success`) used to record whether the function execution is successful (`true`) or not (`false`). The textual specification of the function is self-explanatory.

Concerning a reference implementation, our running example is based on the OpenZeppelin[5] of the ERC20 standard. The Solidity code snippet for the transfer function is presented in Listing 1.1.

```
1  contract ERC20 is IERC20 {
2   /* [...] */
3   mapping (address => uint256) private _balances;
4
5   function transfer(address to, uint256 amount) public virtual override
        returns (bool) {
6      address owner = _msgSender();
7      _transfer(owner, to, amount);
8      return true;
9   }
10  function _transfer(address sender, address recipient, uint256 amount)
        internal {
11     require(sender != address(0), "ERC20: transfer from the zero address");
12     require(recipient != address(0), "ERC20: transfer to the zero address");
13
14     _balances[sender] = _balances[sender].sub(amount, "ERC20: transfer amount
           exceeds balance");
15     _balances[recipient] = _balances[recipient].add(amount);
16     emit Transfer(sender, recipient, amount);
17  }
18 }
```

Listing 1.1. OpenZeppelin implementation for the transfer function

[4] https://eips.ethereum.org/EIPS/eip-20

[5] https://raw.githubusercontent.com/OpenZeppelin/openzeppelincontracts/19c74140523e9af5a8489fe484456ca2adc87484/contracts/token/ERC20/ERC20.sol

A contract in Solidity allows the declaration of types, attributes and functions. In our example fragment, we show only one attribute, `_balances` (Line 3): a mapping from addresses (represented by a 160-bit number) to token balances represented by unsigned (256-bit) integers. The function `transfer` (Line 5) transfers a token amount (parameter `amount`) from the caller (whose address is stored in an implicit argument `msg.sender`) to a destination address (parameter `_to`). This function yields a boolean value that states whether the execution was successful. Note that, despite the differences in some parameter names, the signature of this function is clearly the same as that in the fragment ERC20 EIP.

In the first line of the function implementation, the caller address `msg.sender` is yielded by the (private, internal) function `_msgSender()` that is imported from another Solidity contract; its definition is straightforward and omitted here. This address is saved in the local variable `owner`. Next, `transfer` delegates the actual implementation via a call to the internal function `_transfer` that has an additional parameter to those of `transfer` to record the address of the caller. Finally, the `true` value is returned to indicate the function's successful execution.

The applicability of a function can be captured using the `require(condition)` statement. If the `condition` holds, the execution proceeds normally; otherwise, the function execution aborts and the state before the start of the function execution is preserved. The two `require` clauses of `_transfer` impose that the addresses of both the sender and of the recipient must not be the zero address. These clauses include respective error messages. The next two assignments capture the effect of the transfer: the balance of the sender is decreased by `amount` and that of the destination is increased by the same amount. These statements make use of the functions `sub` and `add`, for integer subtraction and addition; `sub` throws an error message if the result is less than zero, and `add` avoids integer overflow by limiting the sum to the highest integer (2^{256}). The `emit` keyword is used to communicate an event in Solidity; in our example, it is used to log the transfer transaction.

A formal DbC specification in the solc-verify notation for the function `transfer` is given in Listing 1.2.

```
1 /// @notice postcondition (_balances[msg.sender] == __verifier_old_uint(
      _balances[msg.sender]) - amount && msg.sender != to ) || (_balances[msg.
      sender] ==  __verifier_old_uint(_balances[msg.sender]) && msg.sender ==
      to)
2
3 /// @notice postcondition (_balances[to] == __verifier_old_uint(_balances[to
      ]) + amount  && msg.sender  != to ) || (_balances[to] ==
      __verifier_old_uint(_balances[to] ) && msg.sender == to)
```

Listing 1.2. Postconditions for the `transfer` function using the solc-verify syntax

It includes two postconditions (`@notice postcondition`) that are implicitly conjoined. The first postcondition determines that the sender's balance should be decreased by the transferred amount, provided that the sender is not the recipient; otherwise, the balance must be preserved. For an attribute x, the expression `__verifier_old_uint(x)` holds the value of `x` at the start of the function execution. The notation for conjunction (`&&`), disjunction (`||`) and negation (`!`) are standard. The second postcondition states that the recipient's balance must

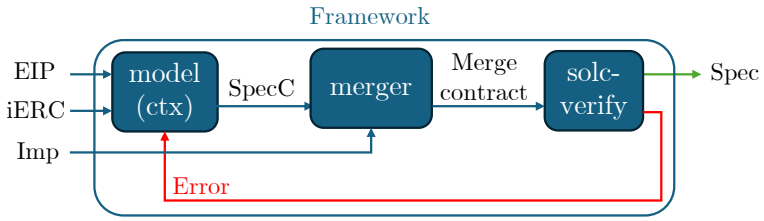

Fig. 2. Framework architecture overview.

increase by the transferred amount, unless the sender is also the recipient, in which case the balance must be preserved.

In the following sections, we use this running example to illustrate the application of our DbC-GPT framework for generating formal specifications and verifying them using solc-verify.

3 Framework

Our framework's architecture is given in Figure 2. It takes as input a specification of an Ethereum standard in the form of a textual EIP (represented by EIP), an annotated ERC interface (iERC), and a reference implementation in the form of a Solidity smart contract, represented by Imp. Given these inputs, the framework generates a DbC specification denoted by Spec.

Internally, the framework is parameterised by an LLM model and a *demonstration context*; the latter consists of some examples of output (i.e. reference specifications) that should help the model in finding the appropriate specification. These elements are combined with a smart contract verifier (solc-verify) in a *counterexample-guided generation process*. The candidate specification (SpecC) generated (given the model and learning context defined) is checked for the syntactic and semantic validity by solc-verify. Semantic validity concerns whether the given reference implementation conforms to the specification in the usual DbC context, but restricted to partial correctness. If solc-verify does not show conformance, the error witnessed is returned to the model (or rather added to the learning context) that generates a new candidate specification. This loop continues until a valid specification is generated or a pre-determined number of iterations is reached. If a valid specification is generated, it is returned to the user of the framework. In the following, we detail this process by describing our methodology to create the *main prompt* (which encompasses the demonstration context) passed to our model, and how our loop is implemented. The model we use in this paper is ChatGPT's GPT-4o[6].

Our framework relies on a property of LLMs, such as ChatGPT, called *in-context learning* (sometimes also referred to as few-shot learning) [8], that is, such models have been shown to perform well when they are given a *learning context*

[6] https://openai.com/index/hello-gpt-4o/

that guides their task-solving process. For instance, given a few examples (i.e. demonstrations) on how the task can be solved for other instances before asking for the solution of the specific instance desired. A concrete example that is often used is to provide many translations of words from one language to another, before asking for the translation of the specific word of interest [8]. We point out that this learning context provided is not part of the model's training (i.e. it does not affect its internal weights) instead it helps the (pre-trained) model to more precisely navigate its "search space" when looking for the correct solution to the task. Both our demonstration context and our counterexample-guided process are designed to make use of this property.

3.1 Main prompt preparation

The main prompt is the starting point of the learning context in our framework, namely, it alone represents the initial learning context given to the model to generate the first candidate specification. The preparation of this prompt involved an ad-hoc *prompt engineering* process[7] where we iteratively refined prompts until we reached the version we present here. In this section, we also illustrate some lessons learned in this prompt engineering process.

The main prompt is formed of three sections: (i) a fixed general description of (i.e. the framing) the task, (ii) a number of examples of outputs that should guide the model in finding the appropriate specification (i.e. *the demonstration context*), and (iii) the inputs (EIP and annotated ERC interface) for the specific generation task we want the model to accomplish.

The first (framing) section is given by Listing 1.3. In the first paragraph, the prompt generically captures the task of generating a formal specification from an ERC interface and an EIP textual file. Note that we point out that only postconditions are to be generated. We also add some instructions and guidance to help the model find the right kind of specification. We instruct the model not to generate function implementations (Line 5), and what is the syntax of a solc-verify postcondition (Line 6) and its positioning with respect to the function for which it is expected to generate the specification (Line 7). Moreover, we have also added some guidance on the generation of the postconditions; they provide some hints as to what semantic aspects the postcondition should capture. In our initial attempts to create a prompt for this task, the ChatGPT model would generate a specification with function implementations or preconditions. Both of these things are not allowed in our specification so we had to refine our initial prompt to reflect these requirements.

The second section of our main prompt is the demonstration context; it gives examples of (i.e demonstrates) reference specifications for ERCs. This section is parameterised by a (possibly empty) sequence of ERCs. They represent demonstrations of expected outputs (i.e. valid specifications) that should help guide the GPT model in its search process. The reference specification of each ERC in this sequence is added, in the order given, as illustrated in Listing 1.4; it exemplifies

[7] https://platform.openai.com/docs/guides/prompt-engineering

```
1  Given an examaple of ERC interface, the ERC interface to be annotated and an
       EIP markdown, generate a specification for the ERC interface with solc-
       verify postconditions annotations, just postconditions, no other
       annotations types, this is very important!
2
3          Instructions:
4
5          - Function Bodies: The specification must not contain function
               implementations.
6          - Postconditions Limit: Each function must have at most 4
               postcondition (/// @notice postcondition) annotations above the
               function signature. Do not exceed this limit under any
               circumstances.
7          - Position: add the solc-verify annotation above the related function
               , example:
8              /// @notice postcondition supply == _totalSupply
9              function totalSupply() public view returns (uint256 supply);
10         - Output format: return the annotated interface inside code fence
               (```) to show the code block. RETURN JUST THE CONTRACT ANNOTATED,
                NOTHING MORE.
11
12         Guidance for Generating Postconditions:
13
14         - State Changes: Reflect how state variables change. For example,
               ownership transfer should reflect changes in token ownership and
               balances.
15         - Conditions on Input: Consider how inputs affect the state variables
               .
16         - Reset Conditions: Ensure certain variables are reset after the
               function execution, if applicable.
```

Listing 1.3. General task description section.

the case of the sequence $\langle$`ERCA`, `ERCB`$\rangle$ where `$reference_specification_ERCA$` and `$reference_specification_ERCB$` are placeholders for where the reference specifications of these ERCs would be placed. For the empty sequence, this section of the prompt is omitted. Again, the addition of these demonstrations follows the principle that GPT models are few-shot learners. During our prompt engineering process, we have observed that passing EIPs in addition to reference specifications in this demonstration section was not helpful as it was hindering the specification search process.

```
1          ERC interface example:
2          ```solidity
3              $reference_specification_ERCA$
4          ```
5
6          ERC interface example:
7          ```solidity
8              $reference_specification_ERCB$
9          ```
```

Listing 1.4. Demonstration context section for ERC sequence $\langle$`ERCA`, `ERCB`$\rangle$.

The last section defines the inputs for the specification generation process, namely, it instantiates the specification generation task. Of course, this part also varies. Listing 1.5 illustrates how this section is constructed for a given ERC; the placeholder `$ERC_annotated_interface$` (Line 6) denotes where the annotated

interface should be placed, `$EIP_name$` (Line 9) where its name should go, and `$EIP_text$` (Line 12) where its EIP textual description should be placed. Line 3 is added only if the demonstration section is non-empty.

```
1  Can you please generate a specification given the following ERC
       interface  (delimited by token ```solidity ```) and EIP markdown
       (delimited by token <eip>)?
2
3  HERE FOLLOWS THE CONTRACT TO ADD SOLC-VERIFY ANNOTATIONS, LIKE THE
       EXAMPLES ABOVE:
4
5  ```solidity
6  $ERC_annotated_interface$
7  ```
8
9  EIP $EIP_name$ markdown below:
10
11 <eip>
12 $EIP_text$
13 </eip>
```

Listing 1.5. ERC input section of main prompt.

In engineering this last section of the prompt, we came across the most interesting prompt refinement. The model was much more sensitive to the EIP's textual specification when it was (additionally) used to annotate the ERC interface. Thus, for each function in the interface, we have extracted the parts of the EIP which are related to it and added as a comment to this function in the ERC interface. For instance, Listing 1.6 illustrates how the `transfer` function is annotated with its corresponding EIP specification. For the specifications that we are interested in this paper, identifying these EIP extracts is fairly simple given that EIPs are already structured in a similar (per function) way. We conjecture that the verbosity of the EIPs might hinder the models ability to identify the text related to a function's specification.

```
1              /**
2               * Transfers `_value` amount of tokens to address `_to`, and
                    MUST fire the `Transfer` event.
3               * The function SHOULD `throw` if the message caller's account
                     balance does not have enough tokens to spend.
4
5               * *Note* Transfers of 0 values MUST be treated as normal
                    transfers and fire the `Transfer` event.
6               */
7               $ADD POSTCONDITION HERE
8               function transfer(address to, uint value) public returns
                         (bool success);
```

Listing 1.6. Example of annotated ERC20 interface for function `transfer`.

3.2 Counterexample-guided specification generation

Once the learning context has been initialised with the main prompt, our framework relies on a counterexample-guided process for the generation of the specification. If the candidate specification generated by our model is not valid, we

return the error information provided by the verifier to the model. More precisely, the error is used to extend and enrich the learning context. In this way, we refine the search for a valid specification.

The specification output by the model consists of a smart contract with function signatures and their corresponding specifications in the solc-verify notation. For checking its validity, we generate a Solidity contract that combines the candidate specification with the reference implementation; we call it a *merge* contract [4]. This combination is carried out by our *merger* component as per Figure 2. Listing 1.7 illustrates an extract of a merge contract where we have combined the implementation (in Listing 1.1) and the specification (in Listing 1.2) of the function `tranfer`. The merge contract brings the annotations that are part of the candidate specification with the function bodies, and auxiliary code, in the implementation. The merge contract (together with auxiliary implementation code) is then checked by solc-verify.

```
1 /// @notice postcondition (_balances[msg.sender] == __verifier_old_uint(
      _balances[msg.sender]) - amount && msg.sender != to ) || (_balances[msg.
      sender] ==  __verifier_old_uint(_balances[msg.sender]) && msg.sender ==
      to)
2 /// @notice postcondition (_balances[to] == __verifier_old_uint(_balances[to
      ]) + amount  && msg.sender  != to ) || (_balances[to] ==
      __verifier_old_uint(_balances[to] ) && msg.sender == to)
3 function transfer(address to, uint256 amount) public virtual override returns
       (bool) {
4     address owner = _msgSender();
5     _transfer(owner, to, amount);
6     return true;
7 }
```

Listing 1.7. Extract of merge contract with the transfer function.

This verifier provides a report which might indicate a syntactic or semantic problem with the implementation, or that it satisfies the specification. In the latter case, the candidate specification has been proved valid and so it is output by the framework. In the former case, we append the output of solc-verify to the *reinforcing* prompt given in Listing 1.8. This prompt simply restates some of the instructions given in the initial prompt — once more, these extra instructions were added due to prompt engineering.

```
1 Instructions:
2     - Function Bodies: The specification must not contain function
          implementations.
3     - Postconditions Limit: Each function must have at most 4 postcondition (
          /// @notice postcondition) annotations above the function signature.
          Do not exceed this limit under any circumstances.
4     - Position: add the solc-verify annotation above the related function,
          example:
5         /// @notice postcondition supply == _totalSupply
6         function totalSupply() public view returns (uint256 supply);
7     - Output format: return the annotated interface inside code fence (```)
          to show the code block. RETURN JUST THE CONTRACT ANNOTATED, NOTHING
          MORE.
```

Listing 1.8. Reinforcement prompt passed back to model.

Listing 1.9 illustrates the kind of output of the solc-verify tool can generate. It presents a syntactical error in the specification generated for the function

`totalSupply`: the model has generated a postcondition that uses the variable `supply` but this variable does not represent either a member variable in the contract or a parameter of the function.

```
1 Error while running compiler, details: Warning: This is a pre-release
      compiler version, please do not use it in production.
2
3 ======= Converting to Boogie IVL =======
4
5 ======= ./solc_verify_generator/ERC20/imp/ERC20_merge.sol ======= Annotation
      :1:1: solc-verify error: Undeclared identifier. supply == _totalSupply
      ^----^
6
7 ======= ./solc_verify_generator/ERC20/imp/IERC20.sol =======
8
9 ======= ./solc_verify_generator/ERC20/imp/math/SafeMath.sol ======= ./
      solc_verify_generator/ERC20/imp/ERC20_merge.sol:38:5: solc-verify error:
      Error(s) while translating annotation for node function totalSupply()
      public view returns (uint256) { ^ (Relevant source part starts here and
      spans across multiple lines).
```

Listing 1.9. Extract of counterexample passed back to ChatGPT.

The prompt combining the reinforcement of instructions with the output of solc-verify extends the learning context provided to the model in the search of a specification; the invalid specification that triggered the creation of this prompt is also part of this context. The next iteration of our loop uses the extended context to generate a new specification, and this process repeats until a parameterised number of iterations is reached. At that point, our framework stops trying to generate a specification.

It is worth mentioning that this counterexample-guided process based on an implementation biases, to some extent, the search for a specification. Ideally, one would look for the most general (weakest) specification capturing the formal properties for a given contract interface and the respective textual specification. However, when the search is driven (even considering the input textual specification) by an implementation, there is a risk that the specification found may be too narrow and applicable only to that single implementation.

Another risk of performing such a kind of search is that the reference implementation may be wrong. In this case, it may drive the generation process away from the textual specification and into a wrong formal specification. For the time being, we do not consider these cases and leave them for future work.

4 Evaluation

For the evaluation of our framework, we consider ERCs 20, 721, 1155. For each of them, we have created a reference specification, and have copied its EIP, its interface, and a popular reference implementation. All these artefacts together with the source code for our tool and scripts to run our evaluation are available in our repository[8]. For our evaluation section, we parameterise our framework with 8 different demonstration contexts. As per Section 3.1, we use sequences of

[8] https://github.com/formalblocks/DbC-GPT

	Input		
Dem. Context	20	721	1155
ϵ	2 [2;4]	1 [9;9]	0
20	10 [0;3]	5 [1;4]	✓2 [8;9]
721	4 [1;5]	10 [0;0]	✓2 [6;9]
1155	5 [1;5]	6 [1;4]	✓10 [0;2]
20, 721	10 [0;7]	10 [0;1]	✓3 [7;9]
20, 1155	10 [0;5]	7 [1;2]	✓10 [0;0]
721, 1155	7 [1;7]	10 [0;4]	✓10 [0;0]
20, 721, 1155	9 [0;4]	10 [0;4]	✓10 [0;2]

Table 1. Evaluation results. For each demonstration context and input ERC, we describe how many times out of the 10 runs a valid specification was generated and the interval on the number of iterations necessary to generate this many specifications.

ERCs to represent the different demonstration contexts that we evaluated. We evaluate all the contexts originating from subsequences of $\langle 20, 721, 1155 \rangle$. The empty sequence is denoted by ϵ.

An evaluation step consists of assessing a demonstration context with an input ERC — the latter means its EIP, annotated ERC interface, and reference implementation. We have evaluated each demonstration context with each input ERC leading to 24 evaluation steps, and we repeat each of these steps 10 times. The repetitions accounts for the fact that LLMs are stochastic models so they might output different results even when presented with the same (input) learning context. Each repetition is a different run of the framework, namely, they start with a fresh learning context, and so there is no "learning" (i.e. context/state sharing) between these runs/repetitions. For this evaluation, we parameterised our framework to try to generate a specification with at most 10 loop iterations. These results are summarised in Table 1. In this table, each cell summarises each of these 10 repetitions by outlining for how many of these repetitions a valid specification was found and, in brackets, the interval on the numbers of iterations necessary to find these specifications. For instance, the cell for demonstration context *20, 1155* and input *721* depicts that out of the 10 repetitions 7 were able to find a valid specification and these runs have taken between 1 and 2 counterexample-guided loop iterations. We use 0 iterations to represent that the framework found a specification with the main prompt alone, that is, without needing even one iteration of our counterexample-guided loop.

Our results show how the demonstration context indeed helps to improve the effectiveness of the generation process. Note, for instance, the significant differences between the runs with an empty demonstration context and the ones with non-empty ones. We point out as well that the intricacies of the specification for the ERC1155 token standard translates into our framework's low effectiveness in generating its specification when compared to the other two standards. This specification is expected to use quantification while the others are not. Finally, note that our results provide evidence for the efficacy of our loop strategy. For

the majority of the runs, the framework needed counterexample-guided loop iterations to find a valid specification.

Note that we have carried out evaluation steps where we try to generate a specification for an (input) ERC that for which the reference specification is being passed in the demonstration context. This may sound like an odd step to analyse but we found it compelling given LLM's stochastic nature and for the purpose of sanity checking (i.e. our framework must generate a specification if it is given the expected reference specification). Note how we have almost complete precision for these cases.

We point out that the ChatGPT 4o model has a limited context, that is, if the number of tokens in the learning context is exceeded the context is truncated. For our evaluation steps, our framework has stayed well within this limit. Hence, this sort of context wrapping is not a problem that we are have encountered or are concerned by at the moment.

We also present graphs that depict, at which iteration, functions in the ERCs have been verified. This per-function analysis allows for a more detailed view in the specification generation process. Figures 3, 4, and 5 depict the per-function analysis for ERCs 20, 721, and 1155, respectively. Broadly speaking, for each function, the corresponding (sub-)plot depicts the frequency in which functions have been verified (y-axis) for a given iteration (x-axis). Each colour in the graph represents a different context that has been used to generated the specification for the given ERC. With these graphs, we can see that, as expected, there seems to exist a co-relation between the complexity of a function and its expected specification and how many loop iterations are necessary for it to be verified. Note, for example, how the `transferFrom`-like functions are more dispersed in these graphs compared to the other functions. Again, this provides additional evidence that our loop-based methodology seems necessary for the generation of specifications for complex functions.

Our results provide encouraging evidence that LLMs can be used to generate formal specification for smart contracts. We have tackled here a class of specifications (i.e. token standards) that are very useful in the domain of smart contracts. Thus, our framework can have already an immediate practical impact in this area. We recognise, however, that this framework and evaluation are only a starting step from which a more thorough investigation should emerge, leading to more robust and general specification-generation frameworks.

5 Related work

The more general aim of our work is to generate formal models from NL requirements. There are two main approaches to using NLP techniques for writing and processing requirements. One is to define a precise grammar to what is usually denoted as a controlled natural language (CNL), using, for instance, the Grammatical Framework [18]. This obviously facilitates NLP, since the requirements can be parsed and then processed in an unambiguous was, but at the expense of imposing a writing style that users must follow. Some examples are [20, 9, 16, 6].

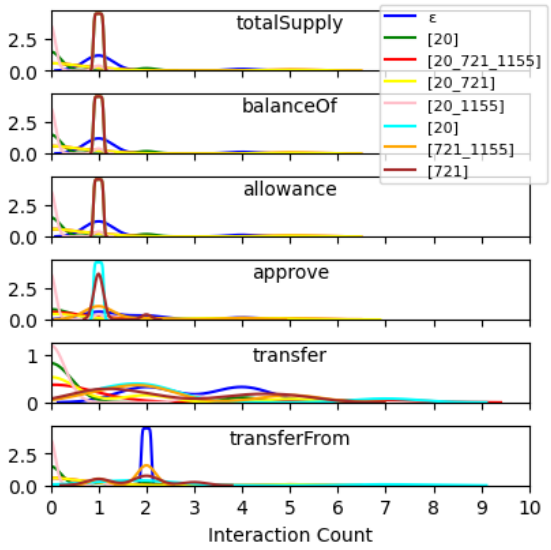

Fig. 3. Per-function analysis for ERC20.

The other approach is to accept as input free NL requirements without imposing any constraints concerning a well-defined grammar. The advantage is that this is clearly more flexible and is likely to address larger contexts; on the other hand, this tends to be significantly more challenging, exactly due to the inherent ambiguity and variability of natural language. Some examples are [2, 19]. We have adopted this more flexible approach, since the intention is to take as input NL specifications of Ethereum standards exactly as they are. Unlike the cited approaches, however, we have based our framework on the use of LLMs.

Unfortunately, there seem to be only a few approaches in this direction. The work in [14] evaluates the precision of a symbolic method and a GPT-based method to generate DbC specifications in the Java Modeling Language (JML) from comments in Java code. We consider here the results related to the GPT-based approach. The GPT model was trained in two stages: a pre-training process on a vast collection of text, followed by a supervised fine-tuning training of the model on a labeled dataset aiming at NLP, but no details of the training are given. Unlike our evaluation, the authors wrote their own (10) NL requirements for their assessment of the models. Although the generated requirements are syntactically and semantically checked, this is not inserted as part of a generation loop. Therefore there is no automatic context improvement from counterexamples, as we do. Out of the 10 requirements, the GPT model could generate only 4 valid JML specifications, although some failed cases were just a matter of missing contextual information.

In [13], the authors investigate the use of ChatGPT to generate loop invariants as annotations for 106 C programs. The authors then used the Frama-C

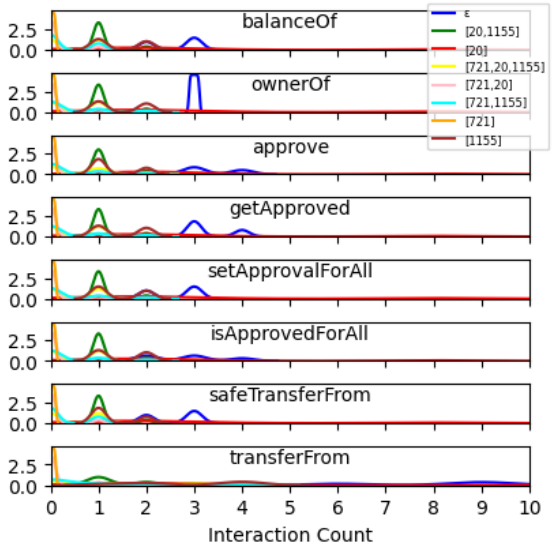

Fig. 4. Per-function analysis for ERC721.

and the CPAchecker verifiers to check the validity (if an invariant holds initially and for each loop iteration), and what they called usefulness (if an invariant helps verifiers in proving tasks). Out of the 106 generated invariants, Chat-GPT managed to generate 75 valid ones; concerning usefulness, Chat-GPT produced 37 useful invariants in comparison with 47 user provided ones. The focus of this work is on reversing engineering formal specifications (particularly loop invariants) from code, whereas we follow a direct engineering approach by generating specifications from NL requirements. Also, despite validity and some form of usefulness checking carried out by the authors, these are not considered in the loop of a(n) (iterative) context improvement as we have done.

6 Conclusion

This paper presented a novel approach to translating natural language into formal specifications for smart contracts based on Large Language Models (LLMs) and, particularly, ChatGPT. The application domain of interest is Ethereum standards, as they are widely used by the blockchain community and have associated textual documentation that we could use, in an unbiased way, to evaluate the DbC-GPT framework we proposed. Our evaluation has shown that it is viable to use ChatGPT for this purpose. Through an iterative process that checks the validity of the generated specifications in the notation of solc-verify, DbC-GPT was able to generate specifications for the 3 standards that were considered, including 8 demonstration context variations. Each variation was formed of a sequence of up-to-three reference specifications for the considered token standards.

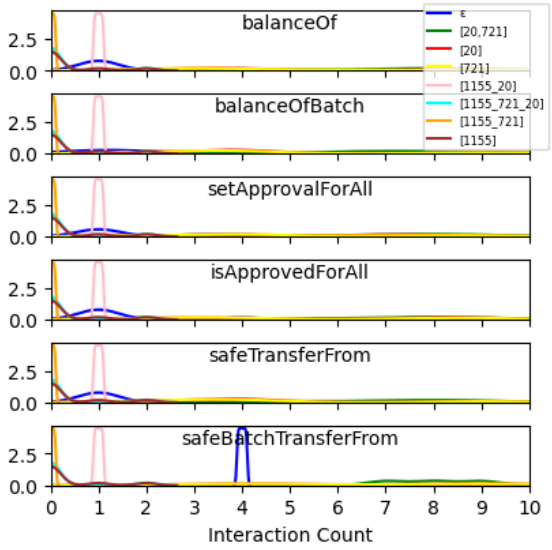

Fig. 5. Per-function analysis for ERC1155.

We were also able to conclude that the more contextual information is provided, the faster a valid specification is generated. More importantly, the iterative process played a crucial role in successfully generating specifications, as, in most cases, valid specifications were obtained only with more than one iteration. As far as we are aware, ours is a first approach to explore this iterative generation of formal specifications from NL descriptions using LLMs.

Looking forward, several opportunities can be explored to enhance and expand the current framework. Our evaluation can be improved to consider many more standards, as well as explore other application domains. The specifications generated by the framework can often be weaker than what is desired; a deeper investigation is necessary and strategies to improve this using appropriate prompt queries is an important topic for future work.

Future work will include using multiple local and non-local LLMs for comparison, such as ChatGPT, Gemini [22], Llama [23], and Claude [21] models, to assess performance differences. We plan to increase the number of experiments and prepare fine-tuning to train models in a standardized manner. Furthermore, we plan to use *a set of implementations* to drive the search for a specification; these implementations can be seen as test cases for the specification being generated, and they should reduce the likelihood of a wrong implementation driving this process to find a wrong specification. Lastly, we will investigate the generation and verification of invariants and preconditions to enhance the robustness and completeness of our framework.

References

1. Solidity language homepage. https://soliditylang.org/, accessed: 2024-05-14
2. Aceituna, D., Do, H., Srinivasan, S.: A systematic approach to transforming system requirements into model checking specifications. In: Companion Proceedings of the 36th International Conference on Software Engineering. p. 165–174. ICSE Companion 2014, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2591062.2591183, https://doi.org/10.1145/2591062.2591183
3. Antonino, P., Ferreira, J., Sampaio, A., Roscoe, A.W.: Specification is law: Safe creation and upgrade of ethereum smart contracts. In: Schlingloff, B.H., Chai, M. (eds.) Software Engineering and Formal Methods. pp. 227–243. Springer International Publishing, Cham (2022)
4. Antonino, P., Ferreira, J., Sampaio, A., Roscoe, A., Arruda, F.: A refinement-based approach to safe smart contract deployment and evolution. Software and Systems Modeling pp. 1–37 (2024)
5. Antonino, P., Roscoe, A.: Solidifier: bounded model checking solidity using lazy contract deployment and precise memory modelling. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing. pp. 1788–1797 (2021)
6. Arruda, F., Barros, F., Sampaio, A.: Automation and consistency analysis of test cases written in natural language: An industrial context. Science of Computer Programming **189**, 102377 (2020)
7. Barza, S., Carvalho, G., Iyoda, J., Sampaio, A., Mota, A., de Almeida Barros, F.: Model checking requirements. In: Ribeiro, L., Lecomte, T. (eds.) Formal Methods: Foundations and Applications - 19th Brazilian Symposium, SBMF 2016, Natal, Brazil, November 23-25, 2016, Proceedings. Lecture Notes in Computer Science, vol. 10090, pp. 217–234 (2016). https://doi.org/10.1007/978-3-319-49815-7_13, https://doi.org/10.1007/978-3-319-49815-7_13
8. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners. Advances in neural information processing systems **33**, 1877–1901 (2020)
9. Carvalho, G., Cavalcanti, A., Sampaio, A.: Modelling timed reactive systems from natural-language requirements. Formal Aspects Comput. **28**(5), 725–765 (2016). https://doi.org/10.1007/S00165-016-0387-X, https://doi.org/10.1007/s00165-016-0387-x
10. Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., R, S., Roy, S.: Program synthesis using natural language. In: Proceedings of the 38th International Conference on Software Engineering. p. 345–356. ICSE '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2884781.2884786, https://doi.org/10.1145/2884781.2884786
11. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M.: CodeBERT: A pre-trained model for programming and natural languages. In: Cohn, T., He, Y., Liu, Y. (eds.) Findings of the Association for Computational Linguistics: EMNLP 2020. pp. 1536–1547. Association for Computational Linguistics, Online (Nov 2020). https://doi.org/10.18653/v1/2020.findings-emnlp.139, https://aclanthology.org/2020.findings-emnlp.139
12. Hajdu, Á., Jovanović, D.: solc-verify: A modular verifier for solidity smart contracts. In: Verified Software. Theories, Tools, and Experiments: 11th International

Conference, VSTTE 2019, New York City, NY, USA, July 13–14, 2019, Revised Selected Papers 11. pp. 161–179. Springer (2020)
13. Janßen, C., Richter, C., Wehrheim, H.: Can chatgpt support software verification? In: Beyer, D., Cavalcanti, A. (eds.) Fundamental Approaches to Software Engineering. pp. 266–279. Springer Nature Switzerland, Cham (2024)
14. Leong, I.T., Barbosa, R.: Translating natural language requirements to formal specifications: A study on gpt and symbolic nlp. In: 2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W). pp. 259–262 (2023). https://doi.org/10.1109/DSN-W58399.2023.00065
15. Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: Verisolid: Correct-by-design smart contracts for ethereum. In: Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23. pp. 446–465. Springer (2019)
16. Nogueira, S., Sampaio, A., Mota, A.: Test generation from state based use case models. Formal Asp. Comput. **26**(3), 441–490 (2014). https://doi.org/10.1007/s00165-012-0258-z, http://dx.doi.org/10.1007/s00165-012-0258-z
17. Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.: Verx: Safety verification of smart contracts. In: 2020 IEEE symposium on security and privacy (SP). pp. 1661–1677. IEEE (2020)
18. Ranta, A.: Grammatical framework. J. Funct. Program. **14**(2), 145–189 (mar 2004). https://doi.org/10.1017/S0956796803004738, https://doi.org/10.1017/S0956796803004738
19. Santiago Júnior, V.A.D., Vijaykumar, N.L.: Generating model-based test cases from natural language requirements for space application software. Software Quality Journal **20**(1), 77–143 (mar 2012). https://doi.org/10.1007/s11219-011-9155-6, https://doi.org/10.1007/s11219-011-9155-6
20. Selway, M., Grossmann, G., Mayer, W., Stumptner, M.: Formalising natural language specifications using a cognitive linguistic/configuration based approach. Information Systems **54**, 191–208 (2015). https://doi.org/https://doi.org/10.1016/j.is.2015.04.003, https://www.sciencedirect.com/science/article/pii/S0306437915000630
21. team, C.: Claude: A next-generation ai assistant by anthropic, https://www.anthropic.com/claude, accessed: 2024-07-19
22. Team, G., Anil, R., Borgeaud, S., Wu, Y., Alayrac, J.B., Yu, J., Soricut, R., Schalkwyk, J., Dai, A.M., Hauth, A., et al.: Gemini: a family of highly capable multimodal models. arXiv preprint arXiv:2312.11805 (2023)
23. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al.: Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288 (2023)
24. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Advances in neural information processing systems **30** (2017)
25. Wang, C., Pastore, F., Goknil, A., Briand, L.C., Iqbal, Z.: Umtg: A toolset to automatically generate system test cases from use case specifications. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 942–945. ESEC/FSE 2015, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2786805.2803187, http://doi.acm.org/10.1145/2786805.2803187

26. Wang, Y., Lahiri, S.K., Chen, S., Pan, R., Dillig, I., Born, C., Naseer, I.: Formal specification and verification of smart contracts for azure blockchain. arXiv preprint arXiv:1812.08829 (2018)