Automated Generation of Code Contracts: Generative AI to the Rescue?

Sandra Greiner
University of Southern Denmark
Odense, Denmark

Noah Bühlmann
University of Bern
Bern, Switzerland

Manuel Ohrndorf
University of Bern
Bern, Switzerland

Christos Tsigkanos
University of Athens
Athens, Greece

Oscar Nierstrasz
Feenk
Bern, Switzerland

Timo Kehrer
University of Bern
Bern, Switzerland

Abstract

Design by Contract represents an established, lightweight paradigm for engineering reliable and robust software systems by specifying verifiable expectations and obligations between software components. Due to its laborious nature, developers hardly adopt Design by Contract in practice. A plethora of research on (semi-)automated inference to reduce the manual burden has not improved the adoption of so-called code contracts in practice. This paper examines the potential of Generative AI to automatically generate code contracts in terms of pre- and postconditions for any Java project without requiring any additional auxiliary artifact. To fine-tune two state-of-the-art Large Language Models, CodeT5 and CodeT5+, we derive a dataset of more than 14k Java methods comprising contracts in form of Java Modeling Language (JML) annotations, and train the models on the task of generating contracts. We examine the syntactic and semantic validity of the contracts generated for software projects not used in the fine-tuning and find that more than 95% of the generated contracts are syntactically correct and exhibit remarkably high completeness and semantic correctness. To this end, our fully automated method sets the stage for future research and eventual broader adoption of Design by Contract in software development practice.

CCS Concepts: • **Software and its engineering** → *Software verification and validation*; • **Computing methodologies** → **Artificial intelligence**.

Keywords: Design by Contract, Software Verification, Generative AI, Large Language Models

ACM Reference Format:
Sandra Greiner, Noah Bühlmann, Manuel Ohrndorf, Christos Tsigkanos, Oscar Nierstrasz, and Timo Kehrer. 2024. Automated Generation of Code Contracts: Generative AI to the Rescue?. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '24), October 21–22, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3689484.3690738

GPCE '24, October 21–22, 2024, Pasadena, CA, USA

ACM ISBN 979-8-4007-1211-1/24/10
https://doi.org/10.1145/3689484.3690738

1 Introduction

Code contracts are widely recognized as a lightweight technique to achieve software reliability and robustness [40]. Introduced in the context of object-oriented programming, the paradigm of *Design by Contract* encourages developers to formally specify code contracts to be automatically verified during program execution [41]. Preconditions specify the expectations for clients invoking a method, while postconditions define the obligations guaranteed by the supplier (i.e., the method). Class invariants ensure the integrity of objects and must hold between consecutive method calls. Consequently, code contracts specify the behavior of collaborating objects and, thus, of the software in a declarative way.

Code contracts enable benefits ranging from preventing, detecting, and semi-automatically correcting errors at all software development stages [26, 40, 41], to an up-to-date documentation facilitating program understanding, maintenance, and evolution [40, 41, 48]. For instance, an extensive empirical study [15] found that whenever projects employ contracts, the contracts tend to be extended over time, most likely as contracts are perceived to be valuable. Despite the benefits, most object-oriented software in practice lacks explicit contracts [4, 11, 15, 52]. Software defined in programming languages featuring native support for Design by Contract, such as Eiffel [42], tend to use more, but usually incomplete contracts [4, 15, 60]. Besides annotation burden, missing tool support, and lack of training [52], the overhead of specifying contracts may be assumed to outweigh its expected return on investment [11],

To mitigate the burden of specifying contracts manually, researchers have proposed (semi-)automated contract inference techniques of three main types: (i) *Static* techniques [8, 20, 21, 28, 37, 45] infer contracts from the source code through static analysis. (ii) *Dynamic* techniques [1, 13,

14, 60] exploit various data obtained from executing programs. (iii) *Mixed* techniques [19, 32] employ auxiliary artifacts, such as natural language specifications. Static inference (i) requires almost no upfront investment but only deduces simple properties that hardly yield comprehensive contracts [15]. Dynamic techniques (ii) may infer contracts of higher completeness but require to execute the programsand to collect the necessary data. Still, achieving completeness is generally infeasible, similar to the task of software testing [12]. While mixed techniques (iii) increase comprehensiveness of code contracts by regarding domain knowledge beyond the source code or execution data, they burden developers to provide the auxiliary artifacts. Despite considerable research efforts and obvious benefits, a recent empirical study on contract usage in the wild [11] found no evidence for an increasing adoption of contracts.

In this paper, we aim to exploit the strengths of a static approach while avoiding its limitations: We envision to statically generate *correct* and *complete* contracts [1] from source code without any upfront investment or auxiliary data. Encouraged by the remarkable potential of Generative Artificial Intelligence (AI), particularly, of Large Language Models (LLMs), in aiding software engineering [17, 24, 44], including code-related tasks [5, 59, 63], we explore whether LLMs can generate high-quality code contracts and examine the following research questions:

RQ 1: *How suitable are existing real-world code contracts to fine-tune an LLM for contract generation?*
We examine the properties and size of data necessary to refine existing LLMs to perform the contract generation.

RQ 2: *Can the fine-tuned LLM generate well-formed code contracts?*
We examine whether the created contracts are syntactically correct and compile without any further manual repairs. This indicates the degree of automation achieved.

RQ 3: *Can the fine-tuned LLM generate logically valid code contracts?*
We examine whether the created contracts are *complete* (i.e., they forbid any unintended behavior) and *correct* (i.e., they allow any known intended behavior).

To answer these questions, in an exploratory study, we examine the quality of code contracts generated by fine-tuned LLMs. We target Java as a widely used object-oriented language. Thus, we need a dataset of real-world contracts suitable for training the downstream task. We collect Java methods featuring code contracts specified as Java Modeling Language (JML) [30] annotations. Based on this novel dataset, we fine-tune CodeT5 [59] and CodeT5+ [58], LLMs pretrained on code-related tasks, to generate code contracts, and we systematically evaluate the models' performances.

As result, we find 73 projects involving about 14k JML annotations by searching multiple open-source repositories (e.g., GitHub and Bitbucket); contributing a novel dataset of JML-annotated software. Despite the resulting, relatively small dataset for fine-tuning, we achieve remarkably good results. The *quantitative* assessment of the models' performances in terms of state-of-the-art NLP metrics (RQ1) are similar to other language translation tasks, notably between natural languages, where LLMs are recognized as state-of-the-art. Second, the well-formedness analysis (RQ2) shows that more than 95% of the generated contracts do not violate the syntax and static semantics of JML; i.e., the created contracts *compile* for software projects neither involved in the fine-tuning nor in the pre-training. By further analyzing the non-compiling contracts, we find recurring *error patterns* mainly related to JML-specific functionality besides omitted symbols. Third, we find that the majority of the generated contracts are logically valid (RQ3); i.e., they are correct and of higher completeness (i.e., prohibit more unintended behavior) than reference contracts.

In summary, this paper contributes:

- A novel dataset of 14k Java methods annotated with JML.
- An instance of LLM-based contract generation using fine-tuned CodeT5 and CodeT5+ models.
- A quantitative analysis of the fine-tuned models' performance using standard NLP evaluation metrics.
- A qualitative analysis of the compilability and logic validity of the generated contracts in software projects, not used for the fine-tuning.
- A replication package [22], including the novel dataset, the fine-tuned models, and the scripts used in our study.

2 Background

This section recalls background information on Design by Contract, before it presents key concepts of Generative AI.

2.1 Design by Contract

Design by Contract [40, 41] introduces *code contracts* as a way to specify expectations and obligations of a software component with respect to its clients. Contracts specify these mutual responsibilities in terms of *preconditions*, *postconditions*, and *invariants* to ensure correctness, clarity, and maintainability. Such contracts are sometimes called lightweight or runtime contracts [15] to distinguish them from strictly formal methods of compile-time verification, such as Hoare logic [23]. While several (technical) domains adopt the contracts paradigm [9, 18, 57], in this work, we remain in its traditional application of object-oriented software development [41]. In more detail, code contracts include:

Preconditions are predicates that must hold before executing a program unit. In object-oriented languages, they typically express the requirements that invokers of public methods should fulfil.

Postconditions are predicates that must hold after executing a program unit. In object-oriented languages, they

```
1 public class Stack<T> {
2   public T top() {...}
3   public boolean isEmpty() {...}
4   //@ requires !(this.isEmpty());
5   public T pop() {...}
6   //@ ensures !(this.isEmpty()) &&
      this.top()==e;
7   public void push(T e) {...}
8 }
```

Listing 1. Java Class `Stack` with JML code contracts.

typically express the obligations or guarantees of the suppliers of public methods.

Invariants are predicates that express the valid states of a program unit. In object-oriented languages, they typically express properties of classes that must hold for each existing instance at the start and end of every public method.

Implementing Design by Contract varies across programming languages. While the programming language Eiffel is the only current language with built-in support to specify code contracts [42], mainstream object-oriented languages support code contracts through dedicated annotations. In our study, we examine code contracts specified in JML [30, 31], expressed as built-in Java annotations. While JML encompasses several keywords and operators, we focus on contracts for Java methods, in the form of pre- and postcondition annotations (`//@ requires` and `//@ ensures`, respectively) .

Listing 1 presents an excerpt of a Stack implementation annotated with JML contracts (`//@`). The method `pop()` specifies as a precondition that the stack not be empty when it is called. The method `push()` specifies as postcondition that the stack will not be empty and the element on top of the stack will be the input object of the method after its execution.

2.2 Large Language Models

Language modeling models the probability of word sequences to predict the probability of future (or missing) tokens. Large Language Models (LLMs) have been pre-trained on immense amounts of text data to learn at a large and general scale.

The Transformer architecture [56] lays the foundation for nearly all recent work on LLMs. Encoder-only models [6, 10, 36] excel in analytical tasks that require a deep understanding of the input text, such as code retrieval [25] whereas decoder-only models [50] excel in generative tasks, such as code creation. Encoder-decoder models [33, 51, 54] excel in adaptive tasks that analyze *and* generate text. Encoder-decoder models include MASS [54], BART [33], and T5 [51]. State-of-the-art encoder-decoder models for code-related tasks encompass CodeT5 [59] and CodeT5+ [58] which are pre-trained on the source code tasks: code summarization, translation, and generation. Due to the encoded knowledge, we explore their capability of annotating Java methods, which requires analyzing the method and generating respective code contracts.

3 Methodology

LLMs pre-trained on general text-to-text tasks require fine-tuning to learn specific downstream tasks. Thus, our study involves three main steps: First, for learning the task of generating code contracts, we *gather* an appropriate *dataset* from open-source repositories (Sec. 3.1). Second, the data serves to *fine-tune* the CodeT5 and CodeT5+ models (Sec. 3.2). Third, we *evaluate* the fine-tuned models' performance with standard NLP metrics quantitatively (Sec. 3.3) and examine the well-formedness of the generated contracts (Sec. 3.4 and Sec. 3.5). We provide the gathered and processed data in an open-source replication package [22].

3.1 Dataset Collection

Noting the absence of previous work on the intersection of Design by Contract and machine learning, the first objective of our work addresses curating a suitable dataset for the training task. While empirical studies on contract usage in practice exist, none of the existing datasets serves for fine-tuning. Estler et al. [15] mainly study how code contracts change as part of software evolution. Their datasets comprise project histories, including lots of redundant information of the unchanged (parts of) contracts. Dietrich et al. [11] examine the usage of contracts in the wild. Their dataset contains several projects not using contracts at all. Schiller et al. [52] conduct case studies involving developers. Their dataset is of small size and only involves Microsoft Code Contracts. Zhang et al. [64] provide a dataset consisting of 514 Python functions from student exercises that the authors annotated with contracts manually, a too small size for fine-tuning a LLM successfully.

Consequently, we curate a dataset tailored to fine-tune the LLMs. This requires a critical number of pairs of source code and respective contracts from diverse projects. Targeting an object-oriented language which is widely used in practice, we search for Java source code annotated with OpenJML specifications, as popular technique for adding code contracts in Java [11, 30, 31, 43]. As JML is mainly used to specify pre- and postconditions for methods [43], we restrict our dataset to JML pre- and postconditions specified for Java methods[1]. We exclude interfaces because they do not contain the actual method body based on which we fine-tune the pretrained CodeT5 models. As a result, an entry in our dataset consists of two parts: (1) The source code of a Java method, and (2) its OpenJML pre- and/or postcondition annotations.

To gather publicly available software projects containing JML annotations, we employed Sourcegraph[2], a code search

[1]Java assertions are not considered as it cannot be reliably determined whether an assertion is used as a code contract or for another purpose [11].
[2]https://sourcegraph.com/search

```
lang:java type:file count:all //@ requires
lang:java type:file count:all //@ ensures
```

Listing 2. Sourcegraph search queries for dataset collection.

Table 1. Hyperparameters used in fine-tuning.

Hyperparameter	Value
Epochs	10
Batch size	8
Learning rate	5e-5
Learning rate warm-up steps	200
Gradient accumulation steps	8
Weight Decay	0.05

engine indexing open-source content of the source code host platforms, GitHub, GitLab, and BitBucket. Listing 2 shows the two input search strings. These queries return line-based occurrences of the OpenJML pre- and postcondition annotations, `//@ requires` and `//@ ensures`, respectively. Next, we extracted the Java methods corresponding with the gathered pre- and postconditions from the respective project repositories. We converted the original Java source code file into srcML [39]. The srcML representation is fully source code text-preserving (including comments, formatting, and white space) but at the same time allows for structural search of the source code. Finally, we used XPath on the XML-based structure of srcML to extract each OpenJML annotation.

3.2 Fine-Tuning of Selected LLMs

In the next step, we use the gathered dataset to fine-tune LLMs on the task of automatically generating OpenJML contracts for a Java method. We selected the open-source models CodeT5 [59] and CodeT5+ [58] which are pre-trained on code-related tasks in popular programming languages and achieve state-of-the-art performance in all code-related tasks of the CodeXGLUE benchmark [16]. These encoder-decoder models excel in adaptive tasks, such as our task of generating code contracts, which requires analyzing a method's source code and generating respective code contracts.

For fine-tuning both models, we employed the PyTorch-based HuggingFace[3] transformers library. To configure the fine-tuning, we used the training hyperparameters suggested for CodeT5 and CodeT5+ [58, 59] and the PyTorch implementation of the AdamW (Adam with Decoupled Weight Decay) optimization algorithm [38]. Table 1 summarizes the hyperparameter values used.

[3] https://huggingface.co/

3.3 Performance Evaluation of Fine-Tuned LLMs

To evaluate the fine-tuned models' performance (answering **RQ 1**), we apply established experimental practice for standard NLP evaluation in a 5-fold cross-validation setup: The original dataset was randomly shuffled and split into five subsets of equal size. We performed five rounds of separate training for each, CodeT5 and CodeT5+. In each round, one of the five subsets served as hold-out set; i.e., it was not used for training but for testing, using the following metrics:

Exact match compares the generated text with the reference text. It either assumes the values 0 or 1, with a score of 1 meaning that the generated sample exactly matches the reference sample.

Cross-entropy represents the default loss function for most Transformer models, ranging from 0 to 1, with 0 as best score. It encodes the difference between two probability distributions, concretely, the difference between the actual distribution of words and the distribution of the words in the test set.

BLEU [47] originates from evaluating machine translation of natural language, ranging from 0 to 1, with 1 as best score. It combines *n*-gram precision with a brevity penalty for the final score. We compute BLEU with a maximum-order of four; i.e., n-grams up to the size of four are considered.

ROUGE [34] is used for evaluating machine translation and text summarization, ranging from 0 to 1, with 1 as best score. We compute ROUGE-L [35], which describes the similarity between the reference and the generated text based on the longest common subsequence.

TER [53] originates from evaluating machine translation. TER describes the number of steps to transform generated text into reference text. The normalized value ranges from 0 to 1, with 0 as best score.

3.4 Syntactic Analysis of Generated Code Contracts

One prerequisite for developers perceiving generated contracts as useful is that the JML annotations are well-formed regarding JML's syntax and static semantics, as motivated by **RQ 2**; i.e., the generated contracts should *compile* without errors. Thus, we investigate whether the generated contracts compile in selected case studies, which were not used in the fine-tuning, and classify the occurring types of errors.

We used the best-performing fine-tuned model of the 5-fold cross-validation to generate contracts for open-source Java projects which are not included the fine-tuning dataset. We implemented a fully-automated application which extracts all Java methods from a given project. The program provides the extracted methods to the fine-tuned model to generate the JML annotations and integrates the generated annotations above the respective method heads in the original source code.

Next, we analyze whether the generated contracts compile with OpenJML [46]. First, we automatically count how many different types of compilation errors are reported by OpenJML for the selected case studies. This provides us with a first indication of the feasibility of using the fine-tuned models. Second, for the contracts violating the OpenJML syntax, one of the authors explored the reasons for the compilation errors reported by OpenJML manually, and examined and classified the errors of the generated annotations. A second author independently controlled the reported results. Eventually all authors discussed and consolidated the qualitative examination and derivation of error classes.

Due to the manual analysis setup of the qualitative examination, suitable subject systems must be tractable for manual inspection. Therefore, we selected two academic Java projects and a sample of well-maintained real-world Java projects: The academic projects, Simple-Stack and Simple-TicTacToe, evolved in an advanced programming course over about 20 years. They allow us to analyze compilation errors through manual inspection. Furthermore, due to their academic nature, the two subjects were neither involved in the pre-training of CodeT5 and CodeT5+ nor did we use them for fine-tuning. They thus provide an initial insight into the models' capability to generalize from the training set. Additionally, we selected the real-world Java projects, Commons-CSV, Jsoup, and Mockito, which we did not use in the fine-tuning. These Java projects are of varying size and complexity and form part of the Defects4J [27] collection of reproducible defects – a well-established and widely recognized set of case studies for software testing. The following list presents all projects selected as subjects (key characteristics thereof are presented in Table 4).

Simple-Stack An academic project designed to introduce Design by Contract in an undergraduate programming course using a classical stack implementation in Java.

Simple-TicTacToe An academic project that implements a command-line version of the popular Tic-Tac-Toe game.

Commons-CSV[4] A library that provides an interface for reading and writing CSV files.

Jsoup[5] A library for parsing, extracting, and manipulating data stored in HTML documents.

Mockito[6] A framework to create mock objects for Java unit testing.

3.5 Logical Analysis of Generated Contracts

In addition to the syntactic analysis, we examine the logical validity of the generated contracts – this entails assessing the intended behavior of the source code equipped with contracts. To reason about the validity of contracts, we examine their (logical) correctness and completeness [1]: Contracts are logically valid if they allow any intended program behavior (*correctness*) but forbid any unintended one (*completeness*). Automatically measuring correctness and completeness of contracts is hardly possible because it requires detailed knowledge of the programs' semantics [1, 15].

To this end, we performed an in-depth manual analysis, comparing generated contracts with a reference implementation with properly defined contracts and well-known program behavior. Simple-Stack and Simple-TicTacToe (Sec. 3.4) served as subjects for this analysis. Both projects were used to introduce Design by Contract academically for decades and, thus, encompass thoroughly engineered contracts. We can reasonably assume that the reference contracts are correct whereas we cannot assume them to be complete – a laborious exercise, contradicting the lightweight formal specification idea of contracts [15]. One author performed the logical analysis and a second one checked it independently.

We classified the generated contracts by comparing them with the pre- and postconditions of the respective reference-contract. Comparing the generated condition c_g with its reference condition c_r results in the following classes:

Equivalent c_g is logically equivalent to c_r (i.e., $c_g \iff c_r$). Generated conditions of this class are correct and can be considered useful for developers.

Weaker c_g is less restrictive than c_r (i.e., $c_r \implies c_g$). Weaker conditions indicate incompleteness because they allow for behavior not intended by the program according to its reference specification.

Stronger c_g is more restrictive than c_r (i.e., $c_g \implies c_r$). Stronger conditions may be (i) valid if they forbid unintended program behavior not restricted by the reference (which may not be complete), (ii) invalid if they forbid intended program behavior which indicates incorrectness of the generated contract.

Unrelated c_g and c_r are logically unrelated. Similar to stronger conditions, they may be valid or invalid (i.e., incorrect).

Please note: an unrelated match may result, for instance, if only one reference precondition exists while our fine-tuned model computes an additional postcondition. If the additional postcondition allows for all intended behavior while forbidding some unintended behavior, it is a valid but an unrelated condition and invalid otherwise.

4 Results

This section presents the outcome of our analyses, guided by the three research questions motivated in Sec. 1.

4.1 RQ 1: How suitable are existing real-world code contracts to fine-tune an LLM for contract generation?

We collected a dataset containing *29,636* JML annotations extracted from *14,270* Java methods implemented in *73* different repositories. Table 2 summarizes key characteristics

[4] https://github.com/apache/commons-csv
[5] https://github.com/jhy/jsoup
[6] https://github.com/mockito/mockito

Table 2. Key characteristics of the 73 repositories in the collected dataset. The age is stated in years.

Metric	JML annotations	commits	forks	stars	pull requests	age
Mean	406	2 520	33	77	56	6.90
Std. Deviat.	2 304	4 956	101	203	148	2.82
Minimum	1	1	0	5	0	0.33
25th perct.	6	21	2	7	0	5.06
Median	15	191	6	17	1	7.22
75th perct.	166	2 275	18	51	26	9.13
Maximum	19 653	21 240	776	1537	919	12.53

Table 3. Performance of the fine-tuned CodeT5, CodeT5+, and CodeT5_without (weka in the dataset) models.

Metric	CodeT5	CodeT5+	CodeT5+ without
Exact match	0.578	0.598	0.439
Cross-entropy	0.046	0.046	0.134
BLEU	0.457	0.456	0.362
ROUGE	0.837	0.849	0.763
TER	0.426	0.427	0.494

of the 73 projects which are all reported in the replication package [22]. We observe that very few projects contribute the majority of annotations. While the median is only 15 annotations per project, the project *weka*[7] contributes almost two-thirds of all annotations. Moreover, we observe that the second to fourth largest projects in terms of the number of contained JML annotations (i.e., *verifast*[8], *verified_SuV*[9], and *OpenJML*[10]) relate to the domain of formal software verification.

As the project weka contributes the majority of annotations in the dataset, we fine-tuned the CodeT5+ model again with a dataset excluding the respective weka-entries. In this way, we explore the influence on the performance while recognizing that the training dataset might be too small for fine-tuning. Table 3 compiles the result of evaluating the fine-tuned CodeT5 and CodeT5+ models in a *5-fold cross-validation* with an 80% train and 20% test split. It reports the measurements of the metrics (cf., Sec. 3.3) for the best-performing model. The first and second column present the results of testing the performance with the dataset including weka whereas the third column captures the results of training without weka.

The results clearly demonstrate that the performance of CodeT5 and CodeT5+ for the large training set is very similar across the evaluation metrics. In fact, we add the third digit to show the marginal differences for the majority of evaluation metrics. More importantly, the fine-tuned models achieve a fairly high exact match score: nearly 60% of the generated code contracts exactly match the original JML annotation. This result also explains the high ROUGE score of about 0.84, indicating a significant textual overlap between the generated annotations and the provided references.

BLEU scores above 0.3 reflect understandable translations whereas BLEU scores above 0.5 reflect good and fluent translation [29]. The achieved BLEU score of about 0.46 measures more than twice as high as the overall performance of CodeT5+ (0.19) in the related code summarization task [58].

The performance of CodeT5+ trained without weka (i.e., with ca. 4k entries) is worse due to the extremely small dataset. Still, the model performs well with an exact match rate of more than 40% and a BLEU score which is just 10% less than achieved with the three times larger dataset. This tendency of about 10% performance reduction can also be observed for the remaining metrics.

> The quantitative performance evaluation demonstrates that real-world contracts are suitable to fine-tune state-of-the-art LLMs pretrained on code-related tasks. Despite the comparatively small dataset, we gain almost 60% exact match rate. Moreover, the performance of the model trained only on about 4k methods shrinks only by about 10% – a remarkable result suggesting that preliminary knowledge about code contracts resides in CodeT5.

4.2 RQ 2: Can the fine-tuned LLM generate well-formed code contracts?

Next, we examined whether the generated code contracts benefit or burden developers by checking whether they can be compiled with OpenJML. Table 4 summarizes the results of our compilation analysis assessing the well-formedness of generated contracts by the best-performing LLM in the training (i.e., CodeT5+ including the weka repository).

The number of generated contracts is higher than the number of methods because the models may generate both pre- *and* postconditions for a single method and even more

[7] https://github.com/svn2github/weka
[8] https://github.com/verifast/verifast
[9] https://github.com/m3mmar/verified_SuV
[10] https://github.com/OpenJML/OpenJML

Table 4. Results of compilability analysis of the generated contracts with OpenJML (ubuntu-20-04-0.17.0-alpha-15).

Project	#methods	#generated contracts	#errors	success rate [%]
Stack	15	21	20	4.80
TicTacToe	30	48	1	97.9
commons-csv	235	333	10	97.0
jsoup	1 288	2 182	36	98.4
mockito	1 770	2 589	175	93.2
		∑ 5173	**∑ 242**	**∅ 95.3**

than one annotation of either type which are counted each. The *#errors* column presents the number of errors counted by OpenJML[11]. The *success rate* computes the ratio between the remaining successfully compiled contract annotations and all generated contracts for a project. Remarkably, more than 95% (weighted average) of all generated annotations compile, thus, they are syntactically correct. The project Simple-Stack represents an outlier where the model performs badly with only one compiling annotation which we discuss further when we analyze the reported errors.

While the model mostly acquired the syntax, semantics and functionality of JML, still about 10% of the generated contracts do not compile. We analyzed the type of reported errors in-depth 1) by counting the types of errors reported by the OpenJML compiler automatically [22], and, 2) by examining the erroneous contracts manually.

First, Table 5 collects the resulting counts of compilation errors for *all* projects. The first column states the error message reported by the OpenJML compiler in condensed form, the second one states the number of occurrences, and the last one the relative frequency among all 242 errors. We do not show errors occurring less than three times but list them together with the error message in the replication package [22].

The majority of reported JML compilation errors are of syntactic nature. Almost half of all errors, such as the illegal start of an expression or statement (76 times) or missing semicolons (27 times), occur in complex annotations where mainly wrong symbols are used to combine clauses. Moreover, the errors partly represent a rather human behavior which may, for instance, forget closing symbols. Particularly, for the Simple-Stack project, the only type of error that the model makes 20 times is stating a protected field in an annotation without assigning a specific visibility annotation to the field – as reported below – one functionality of JML that the model failed to acquire, most likely because such situations were not covered in the training.

[11] OpenJML may report more than one compilation error for one annotation.

Table 5. Topmost occurrences of compilation error types.

Error message	#	amount
illegal start of expression	76	31.4 %
<char> missing or expected	48	19.8 %
unexpected JML token:	47	19.4 %
missing semicolon	27	11.2 %
public access of protected visibility	16	6.6 %
unknown backslash	6	2.5 %
incorrectly formed/terminated statement	4	1.7 %
empty character literal	3	1.2 %

```
1 public class Bucket {
2   //@ spec_public
3   private int size;
4   //@ ensures \result == size;
5   public int getSize() { return size; }
6 }
```

Listing 3. Example of a correctly referenced private variable.

Error Classification. Second, when analyzing the compilation errors manually, one author inspected a random sample from the three real-world projects of about 100 error-provoking annotations and examined all annotations of the academic projects manually. A second author checked the results and they were discussed among all authors. We find that compilability problems exhibit a small number of error patterns, of syntactic, functional, and semantic nature.
First, the model makes purely **syntactic** mistakes:

Omitted symbols: Similar to Java, every statement in JML must end with a semicolon and double quotes must enclose Strings. Many generated contracts miss closing semicolons, String delimiters, or closing or opening brackets, resulting in JML errors.

Second, several mistakes relate to the **functionality and restrictions** of JML and OpenJML:

Visibility violations: In JML, visibility rules prohibit clients seeing a public method declaration and its specification from accessing fields or methods of more restrictive visibility. The model sometimes stated protected or private variables in annotations for public methods, resulting in JML errors. To prevent the error, an additional annotation `spec_public` needs to be stated before the declaration of the private or protected field. Listing 3 presents how such references are stated correctly.

Inheritance violations: Any pre- or postcondition of a method in a class or interface must start with the keyword `also` if and only if this method is declared in the parent type extended or implemented by the current type. Some generated annotations of this type lack the keyword `also`.

```
1 //@ ensures \result == (squaresLeft() > 0) ==>
      (squaresRight() > 0)
2 public boolean notOver() {
3     return this.winner().isNobody()
4     && this.squaresLeft() > 0;
5 }
```

Listing 4. Example of symbol hallucination: The method `squaresRight()` does not exist in the source code.

Finally, the model makes **semantic** mistakes:

Symbol hallucinations: The model generates annotations that involve non-existing variables, objects, or method names. In some cases, the invented symbols are somehow related to other existing symbols. Listing 4 presents an instance of this error. The method `squaresLeft()` computes how many squares of a Tic-Tac-Toe game are yet unmarked. As the method invokes `squaresLeft()`, the model assumes a method `squaresRight()` to exist.

Comparing incompatible types: The trained model uses comparison operators with objects of incompatible types. For instance, a generated precondition states that the result of a method returning a Boolean should be equal to a specific string.

Invalid use of `\result`: In JML, the keyword `\result` accesses the return value of a method inside an annotation, which is particularly used in postconditions. In some cases, the generated contracts contain `\result` for void methods or concatenate the keyword with other method-specific Strings (unknown backslash error), resulting in JML errors.

> More than 95% of the contracts generated by the fine-tuned LLM are well-formed. An in-depth, automatic and manual, qualitative analysis of the remaining erroneous contracts finds that syntactic errors are most often provoked by small and easy-to-fix errors, such as missing closing symbols, e.g., missing semicolons or brackets. Additionally, some static semantic semantics violations result from missing knowledge about the Java method's environment, such as the wrong reference of protected or private fields or wrong typing assumptions.

4.3 RQ 3: Can the fine-tuned LLM generate logically valid code contracts?

To analyze the logical validity (**RQ 3**), we examined the contracts generated by the fine-tuned CodeT5+ model for the subjects Simple-Stack and Simple-TicTacToe manually. While they convey a considerably number of compilation errors, we can perform an unambiguous "quick fix" in all cases. Table 6 summarizes the results of the analysis. It presents the relative frequency of the generated pre- and postconditions with respect to their reference counterparts categorized by the comparison classes (c.f., Sec. 3.5). While possible, valid unrelated contracts do not occur (omitted from table). The full classification is available in our replication package [22].

First, we observe that only about 16% of the generated conditions on average are semantically equivalent to their reference counterparts although we obtained exact match scores of almost 60% in our performance evaluation guided by RQ 1 (c.f., Table 3). A possible reason is that the performance of the fine-tuned CodeT5+ decreases for projects it has not seen before, such as Simple-Stack and Simple-TicTacToe. Additionally, Simple-Stack and Simple-TicTacToe do not contain any trivial getter and setter methods. In our manual analyses, we observed that such boilerplate methods often trigger the creation of simple "non-null checks" for which exact matches are likely. Conversely, the majority of the semantic equivalences demonstrate that the model understood the intended program behavior, while only some are rather trivial. As example of a trivial generated condition, `//@ requires true;` or `//@ ensures true;` were generated in cases of an empty reference precondition. In contrast, for instance, the precondition `//@ requires row >= 0 && row <= gameState.length;` was generated which captures a non-trivial in-range check on board game objects correctly.

Second, less than 10% of the generated conditions are weaker than their reference counterparts. This indicates that the generated contracts achieve a remarkable level of completeness, as the generated contracts forbid unintended behavior almost as strictly as the reference contracts. For instance, in Simple-Stack, the model did not generate a postcondition for the method `push(E item)` although it should ensure that the given `item` resides on top of the stack after execution. As an example of a missing precondition, the model did not generate a contract for the Stack object's method `top()` which may only be invoked on non-empty stacks. This is particularly interesting as the respective precondition `//@ requires size > 0;` was correctly generated for the method `pop()`.

Remarkably, almost two-thirds of the generated contract annotations (summary: 65.2%) are stronger than their reference conditions. The vast majority of them are logically valid as they forbid more unintended program behavior than the reference contracts. While these numbers demonstrate that the generated contract annotations are more complete than the reference contracts, arguably not all of the generated stronger conditions add value for developers. Particularly, the model created many postconditions by just copying the expression of the return statement in the case of Boolean functions or by combining the expression of the return statement and `\result` using an equivalence operator in the case of different return types. Though arguably naive, this strategy creates postconditions which are logically valid, and explains the large fraction of generated postconditions that are stronger than the reference conditions.

Table 6. Relative frequencies of logical validity evaluation of generated contracts compared to their reference annotation.

	Equiv.	Weaker	Stronger (valid)	Stronger (invalid)	Unrelated
Pre-cond.	25.0%	5.0%	45.0%	10.0%	15.0%
Post-cond.	8,7%	13,0%	56,5%	17,4%	4,3%
Weighted Average	16,3%	9,3%	51,2%	14,0%	9,3%

Lastly, only less than 10% of the generated contracts on average are logically unrelated to their reference. Together with the invalid stronger annotations, this indicates that the model reflected the intended behavior for more than 75% of the subject methods, correctly.

First, more than 15% of the generated pre- and post-conditions on average are logically equivalent to their reference contract, constituting correct contracts. More than 50% of the generated conditions are stronger than their reference and logical valid. While the added value of some annotations for developers may be limited, completeness is even higher than for the reference contracts. Only less than 10% of the generated conditions on average are weaker than their reference counterparts. About 10% of the generated contracts are logically unrelated to their reference annotation and invalid.

5 Discussion

This section discusses our insights gained from fine-tuning the two LLMs and using them for automatically generating code contracts for Java, as representative of a mainstream, widely used object-oriented programming language. Furthermore, we outline how to build on the remarkable results of our work to establish the automatic code contract generation through Generative AI in practice.

First, regarding the prevalence and usage of Design by Contract in practice, we can confirm previous research findings concerning JML annotations [11]. Detecting only 73 publicly available projects using JML annotations by mining three of the leading open-source code platforms indicates that Design by Contract is currently not widely applied in Java. While JML and OpenJML are not the only facilities used for Design by Contract, they are commonly used in Java. Thus, it is unlikely that we missed a large number of Java projects due to the focus on JML. Further in-depth examination of the 73 projects showed that the projects containing JML annotations are generally not widespread and often inactive. 47 projects out of the 73 project did not mention a single commit in the last year – an indicator that Design by Contract is not a central focus in current software development. Remarkably, we observe that three out of the four repositories with the highest number of JML annotations implement software relating to software verification. This gives evidence that Design by Contract is particularly prevalent among developers with verification domain knowledge.

By using the gathered dataset of 14k annotated Java methods, we fine-tuned the models CodeT5 and CodeT5+ for the task of contract generation. We also tried to use them without training but found that the models do not understand the task at hand when prompting them. In contrast, after fine-tuning, we obtain notable quantitative results – particularly for the exact match and ROUGE. When assessing those rates, it is important to regard that a program, and in our case code contracts, can be implemented in infinitely many ways to behave equivalently to the reference. Thus, almost two thirds of all generated contracts exactly matching the reference annotation, and more than 80% overlapping in the longest common subsequences is a remarkably good result. While the metrics BLEU, ROUGE, and TER are designed for natural language processing tasks, they are used to evaluate code-related LLMs in related work [16, 58, 59]. Particularly, the BLEU scores of ca. 0.46 indicate understandable translations up to good translations [29], an estimation which is also in line with our qualitative analysis on subject systems not used in the fine-tuning.

Applying the best-performing fine-tuned CodeT5+ model to real-world software projects not used in the fine-tuning results in more than 75% compiling JML annotations on average. By automatically and manually analyzing the compilation errors, we identify visibility violations and misuse of return parameters besides pure syntactic errors, such as missing symbols as the majority of provoking errors. Particularly, for the subject yielding the worst performance, Simple-Stack, we identify that the type of missing annotation could not be provided. It requires annotating fields whereas our fine-tuned model can only annotate methods due to the limited size of the context window which so far restricts the number of input and output tokens processed by the model.

When categorizing the model errors we find syntactical, semantic, and OpenJML-related issues. Particularly for the semantic mistakes, we sometimes observe a type of model behavior resembling the way humans would use natural language. A possible explanation for that behavior is the fact that the underlying foundational model of CodeT5 and CodeT5+, T5 [51], was originally trained on natural language processing tasks. Moreover, we found the model hallucinating and referencing non-existing Java symbols, a phenomenon observed in previous work on LLMs [2]. Such effect

```
1 //@ ensures x == 0 ==> \result == 0;
2 //@ ensures x < 0 ==> \result == -1;
3 //@ ensures x > 0 ==> \result == 1;
4 public static int sign(int x) {
5     if (x == 0) { return 0; }
6     else if (x > 0) { return 1; }
7     else { return -1; }
8 }
```

Listing 5. Correct and complete JML code contract generated by the fine-tuned CodeT5+ model.

may be mitigated providing more context information to the model; e.g., the declared fields of a class, the entire class, or even the entire project. More context information may also solve the problems related to visibility violations in OpenJML. Models with access to fields and their visibility modifiers, may not make such mistakes.

The results of our logical analysis are remarkable. Notably, we observe not only that the majority of the generated contracts are logically correct, but also that the completeness is even higher than of the reference contracts. In some cases, the model captured even non-trivial, intended program behavior and converted it into generated contracts. While we performed the manual logical analysis only on Simple-Stack and Simple-TicTacToe — resulting in a small sample size — we identified *many* cases supporting the high logical validity. For instance, Listing 5 demonstrates a contract generated by our fine-tuned LLM that captures the semantics of the function `sign()` correctly. Conversely, we found *rare* cases where the model generated tautologies as preconditions (e.g., `//@ requires true == true`) or contradictions as postconditions (e.g., `//@ ensures 2 == 1`).

Consequently, our work clearly demonstrates that using a fine-tuned LLM to generate contracts offers a large potential to increase the adoption of code contracts in practice. Without any surplus effort code contracts can be generated and embedded with our method in Java projects. At the current state of our research artifact, developers still have to verify the contracts manually. However, as potential next steps an automated post-processing step could automatically compile and quick-fix frequent errors, such as adding a missing semicolon or closing bracket; respective error classes and types are reported in Sec. 4.2. Furthermore, very recent developments in the AI and the software engineering community could be exploited to increase the context window size to provide further context information which will positively influence the overall quality of the automated approach. If no additional burden is put on developers while ensuring high quality and lightweight verification, code contracts can be expected to be adopted more regularly in software projects.

6 Threats to Validity

This section discusses threats to the internal and external validity [61] of our experiment and evaluation of code contracts generated with the fine-tuned CodeT5+ model.

6.1 Internal Validity

Regarding the gathered novel dataset of 14k Java methods, we did not analyze the quality of all included contracts in-depth but only small random samples. This threatens internal validity because not all annotations in the dataset may compile without errors. As a consequence, the quality of the collected annotations may impact the quality of annotations learned by the model.

Moreover, the original pre-training datasets of CodeT5 or CodeT5+, consisting of billions of code samples, encompasses some projects of our training dataset and includes the three Java projects (commons-csv, jsoup, and mockito) we used as subject systems for the syntactic and semantic analysis. In terms of our training dataset, that means the model may have already seen some JML annotations before the fine-tuning but without learning the task of generating contracts. Even if our training dataset had been seen, it would represent an infinitely small portion of the original training data which might have influenced our results positively. To mitigate this threat, we examined the academic projects Simple-Stack and Simple-Tic-Tac-Toe for which we can also analyze the logic validity of the generated contract annotations.

For assessing the fine-tuned models' contract generation capabilities, a cross-cutting question is whether a suitable baseline exists to which we may compare our results. Despite a large body of research on contract inference (cf., Sec. 7), a fair comparison is difficult. Different techniques approach contract generation with different goals and assumptions. On the one hand, static analysis techniques generally outperform a probabilistic method, such as LLMs, regarding the compilability of the generated contracts, yet at the price of a serious lack of completeness. On the other hand, the quality of dynamic inference or of exploiting auxiliary artifacts heavily depends on the provided data. Thus, the only viable evaluation goal of an experiment would be to analyze and quantify this trade-off. While we envision such comparison in the future, in terms of the scope of this work, we are interested in the feasibility of this radically new approach to contract generation. The only fair baseline to compare with might be another Generative-AI-empowered approach, which to the best of our knowledge, does not exist yet.

For assessing the model performance, we employed the standard metrics, BLEU, ROUGE, and TER, which are designed for assessing natural language tasks. Therefore, their applicability to code-related tasks can be contentious [55]. Still, they can serve as a measure for relative comparisons of different model-generated contracts. To mitigate this threat, we draw our conclusions not only based on the quantitative

performance, but also qualitatively analyze the generated contracts in terms of well-formedness and logical validity in selected subject systems, not used in the fine-tuning.

Finally, we assessed the logical validity of generated contracts by comparing them to a reference specification. This yields a relative rather than an absolute assessment of logical correctness and completeness, which might be biased by flaws in the reference contracts. However, the projects for this kind of analysis were carefully selected to include reference contracts which matured over decades. Thus, we can safely assume the reference contracts to be correct and useful from developers' perspective. The manual character of the analysis may involve human errors, which we minimized by checking the results independently and discussing them among all authors.

6.2 External Validity

In our experiment, we focused on the Java programming language and JML annotations, which is a threat to the generalizability of the results obtained with respect to other contract and programming languages. However, our described methodology can be transferred to such languages, and similar results are expected to be achieved for similar languages.

Furthermore, sampling bias might be an issue that impacts the generalizability of our results for **RQ 1**. One software project (weka) contributes two-thirds of the JML annotations in our dataset. The coding style and guidelines applied in this project likely influenced the way our fine-tuned models understand code and generate code contracts. We mitigate this threat by examining the model behavior when excluding the weka project. Despite the very small training set, the model performance decreased at a comparatively small rate – an indicator that the bias introduced by weka remains low.

Sampling bias might also affect the results of **RQ 2** and **RQ 3**, which are based on a convenience sample for the sake of keeping the highly demanding analyses feasible, particularly, the logical analysis. While the results may not generalize to other projects, we analyzed compilability for representative Java projects heavily used in software engineering research (Defects4j dataset). Furthermore, the academic projects have not been seen by the LLMs, neither in pre-training nor fine-tuning, and thus indicate generalizability.

Lastly, mapping the compilation errors to classes of compilability problems was obtained from investigating more than half of the compilation errors manually. The classification may still be incomplete but remained quite stable even after examining more than half of all errors, providing confidence in their representativeness.

7 Related Work

Our generative approach resides at the intersection of classical contract inference and Generative AI. Thus, we examine both areas below. We consider static, dynamic, and mixed contract generation techniques, focusing on approaches respecting Meyer's notion of a contract [40, 41] as also adopted in our work. Regarding Generative AI in software engineering, we examine the work closest to ours.

Static inference techniques examine the source code of a program to derive properties of interest through analytical methods. Accurately inferring properties, except for the most basic ones, is an undecidable problem in general [60]. Consequently, static techniques focus on tractable problems, favoring correctness over completeness. Available techniques range from abstract interpretation [7], for automatically discovering interval constraints [8] or affine relations [37] among variables of a program, to inferring extended typing information [45], out-of-bound and pointer checks [20], and invariants of loops [21, 28]. While all of these properties may form the basic building blocks of code contracts, they are too simple to capture a program's actual behavior.

Dynamic inference relies on data obtained from program executions, ranging from simple traces to snapshots of the entire object graph of an object-oriented program. Several techniques are based on the seminal work on Daikon [14], an approach and tool for dynamically generating likely program invariants by matching pre-defined patterns of logical and arithmetic relationships. Schiller et al. [52] generate traces of .NET programs compatible with Daikon, a Visual Studio add-in turns the detected invariants into contracts. On top of simple relationships, AutoInfer [60] can infer quantifiers. It performs a template-based invariant detection, similar to Daikon, to infer contract clauses. The clauses are further analyzed to eventually become contract candidates validated against a given test suite. Another work [1] infers so-called visual contracts over object Graph modifications performed by Java operations, formally capturing pre- and postconditions as Graph transformation rules. Contrary to static inference, dynamic inference cannot be applied off-the-shelf: Typically developers need to properly instrument and execute programs. Even with tool support, such as automated test generation, it remains a tedious task prone to errors. To this end, dynamic inference is limited to the behaviors that have been actually executed.

More recent inference approaches involve domain and application knowledge beyond the source code or program execution data to increase the logic validity of code contracts. Milanez et al. [19] propose an approach to automatically generate contracts based on natural language code commentary using supervised learning. Their tool "ContractSuggestor" generates contract-like constructs in AspectJ[12] using tagged Javadoc comments of two types: Non-null properties and relational properties. The technique differs from our approach as it uses a supervised learning classification algorithm. Thus, it is limited compared to a generative language model. Moreover, the algorithm does not use the source code for the

[12] https://eclipse.dev/aspectj/

classification, but only the Javadoc comments. A similar approach [32] takes the source code of a Java program together with a natural language specification as input and derives JML contracts. A pipeline, including tokenization, stopword removal, normalization, and lemmatization is used together with custom lexical rules to convert the contracts given in natural language into JML clauses. Contrary to our work, both approaches [19, 32] require contracts manually specified in natural language which are merely converted into a machine-readable form.

Recently, Generative AI was examined to derive program invariants with similar results to ours [3, 49, 62]. First, examining the baseline of untrained or unspecific prompts does not produce reasonable contracts. Conversely, once properly prompted or trained, the LLMs create more valid contracts than available in the training sets. In more detail, Lemur is a novel tool and framework to perform program verification of loop invariants by using GPT4 [62]. It employs the LLM to generate novel properties which are treated as assumptions up to the point that they are proven to be invariants. Contrary to our work, the approach does not fine-tune the model. It relies on continuously prompting and ranking the results and outperforms existing ML-based verifiers in benchmarks for invariant synthesis significantly. A similar experiment [3] explored prompting GPT3 and GPT4 for ranking invariants generated with zero-shot prompting by the LLMs. Training the model to properly rank the generated invariants improved the ranks. Pei et al. [49] examine scratchpad prompting and fine-tuned GPT4 based on test sets created with Daikon. Both methods outperformed Daikon, particularly, for a small number of available execution traces. In contrast to these works, we fine-tuned T5 pretrained on code related tasks on a novel dataset to generate pre- and postconditions for methods with remarkable results. Moreover, we did not only regard the resulting ranking of the generated valid invariants or the accurate creation of expected invariants, but also examined the contracts generated for unseen subjects qualitatively and quantitatively. In a post-processing step, our uniquely reported error classification may help to optimize the results.

8 Conclusion

This paper presented how to harness the power of LLMs to automatically generate code contracts for Java methods, as representative of a mainstream object-oriented programming language, without depending on dynamic analyses or auxiliary data. While we can confirm earlier findings that Design by Contract is currently not widely adopted by software developers, we could collect a novel dataset of approximately 14k Java methods annotated with almost 30k JML contracts, which – as one contribution of our study – can fuel further research in the field. We used the JML contracts dataset to teach two state-of-the-art LLMs the task of automatic code contract generation. Despite the small dataset, the fine-tuned models achieve remarkable quantitative results when evaluated with traditional NLP metrics. Qualitatively investigating the contracts generated for subjects not used in the fine-tuning demonstrated that more than 95% of annotations compile and the majority are of impressive logical validity.

Thereupon, we identify key research directions towards dataset diversification, expanding context, and input engineering. Firstly, diversifying the dataset — for instance, by including contracts from frameworks other than OpenJML (e.g., oval[13], cofoja[14]) — represents one way to broaden the training. Secondly, more context information, such as including class fields or the interface declaration in the input, may benefit the quality of generated contracts. Thus, prompt engineering and appropriate input slices are future ways to explore for improving the quality of generated contracts.

All in all, our work represents a pioneering step to a novel application area of LLMs, to statically retrieve contracts of high correctness and completeness. In comparison to previous traditional static, dynamic and mixed methods, the correctness and completeness of the *fully automatically* generated code contracts is remarkably high without demanding any additional burden from the software developers. Even for syntactically or logically erroneous contracts, the fixes are rather easy and can be expected to be minimized through automatic postprocessing and methods that allow LLMs to process more information, particularly, entire software repositories to provide necessary context knowledge. In this way, Generative AI provides a highly valuable resource to be exploited for increasing the adoption of the lightweight concept of Design by Contract in software engineering practice.

Acknowledgments

This work is partially supported by the Hellenic Foundation for Research and Innovation Project 15706.

References

[1] Abdullah Alshanqiti, Reiko Heckel, and Timo Kehrer. 2018. Inferring visual contracts from Java programs. *Automated Software Engineering* 25 (2018).

[2] Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, Quyet V. Do, Yan Xu, and Pascale Fung. 2023. A Multitask, Multilingual, Multimodal Evaluation of ChatGPT on Reasoning, Hallucination, and Interactivity. (2023).

[3] Saikat Chakraborty, Shuvendu Lahiri, Sarah Fakhoury, Akash Lal, Madanlal Musuvathi, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking LLM-Generated Loop Invariants for Program Verification. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). ACL, 9164–9175.

[4] Patrice Chalin. 2006. Are practitioners writing contracts? *Rigorous Development of Complex Fault-Tolerant Systems* (2006).

[13] https://sebthom.github.io/oval

[14] https://github.com/nhatminhle/cofoja

[5] Mark et al Chen. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs]

[6] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. arXiv:2003.10555 [cs]

[7] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*.

[8] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program. In *Proc. of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*.

[9] Patricia Derler, Edward A Lee, Stavros Tripakis, and Martin Törngren. 2013. Cyber-physical system design contracts. In *Proc. of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proc. of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Vol. 1*. ACL.

[11] Jens Dietrich, David J. Pearce, Kamil Jezek, and Premek Brada. 2017. Contracts in the Wild: A Study of Java Programs. In *31st ECOOP 2017 (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1–29.

[12] Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall.

[13] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 1999. Dynamically discovering likely program invariants to support program evolution. In *Proc. of the 21st international conference on Software engineering*.

[14] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007).

[15] Hans-Christian Estler, Carlo Alberto Furia, Martin Nordio, Marco Piccioni, and Bertrand Meyer. 2014. Contracts in Practice. In *FM 2014: Formal Methods*. Vol. 8442. Springer, 230–246.

[16] Shuai Lu et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021). https://doi.org/10.48550/arXiv.2102.04664

[17] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. *CoRR* abs/2310.03533 (2023). https://doi.org/10.48550/ARXIV.2310.03533 arXiv:2310.03533

[18] Hakim Ferrier-Belhaouari, Pierre Konopacki, Régine Laleau, and Marc Frappier. 2012. A design by contract approach to verify access control policies. In *IEEE 17th Intl. Conf. on Engineering of Complex Computer Systems*. IEEE.

[19] Alysson Filgueira Milanez. 2018. *Fostering Design By Contract by Exploiting the Relationship between Code Commentary and Contracts*. Ph. D. Dissertation. Federal University of Campina Grande.

[20] Cormac Flanagan and K Rustan M Leino. 2001. Houdini, an annotation assistant for ESC/Java. In *International Symposium of Formal Methods Europe*. Springer.

[21] Carlo Alberto Furia and Bertrand Meyer. 2010. Inferring loop invariants using postconditions. *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday* (2010).

[22] Sandra Greiner, Noah Bühlmann, Manuel Ohrndorf, Christos Tsigkanos, Oscar Nierstrasz, and Timo Kehrer. 2024. *Replication Package: Automated Generation of Code Contracts - Generative AI to the Rescue?* https://zenodo.org/doi/10.5281/zenodo.13351003

[23] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969).

[24] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John C. Grundy, and Haoyu Wang. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. *CoRR* abs/2308.10620 (2023). https://doi.org/10.48550/ARXIV.2308.10620 arXiv:2308.10620

[25] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. arXiv:1909.09436 [cs, stat]

[26] J-M Jazequel and Bertrand Meyer. 1997. Design by contract: The lessons of Ariane. *Computer* 30, 1 (1997), 129–130.

[27] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proc. of the 2014 Intl. Symposium on Software Testing and Analysis*. ACM, 437–440.

[28] Laura Kovács and Andrei Voronkov. 2009. Finding loop invariants for programs over arrays using a theorem prover. In *International Conference on Fundamental Approaches to Software Engineering*. Springer.

[29] Alon Lavie. 2011. Evaluating the Output of Machine Translation Systems. In *Proc. of Machine Translation Summit XIII: Tutorial Abstracts*.

[30] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 2006. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *ACM SIGSOFT Software Engineering Notes* 31, 3 (May 2006), 1–38.

[31] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. 2005. How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification. *Science of Computer Programming* 55, 1-3 (March 2005).

[32] Iat Tou Leong and Raul Barbosa. 2021. Generation of Oracles Using Natural Language Processing. In *2021 28th Asia-Pacific Software Engineering Conf. Workshops (APSEC Workshops)*. IEEE, 25–31.

[33] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-Training for Natural Language Generation, Translation, and Comprehension. In *Proc. of the 58th Annual Meeting of the Association for Computational Linguistics*. ACL, 7871–7880.

[34] Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out*. ACL, 74–81.

[35] Chin-Yew Lin and Franz Josef Och. 2004. Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics. In *Proc. of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*. 605–612.

[36] Zhuang Liu, Wayne Lin, Ya Shi, and Jun Zhao. 2021. A Robustly Optimized BERT Pre-Training Approach with Post-Training. In *Proc. of the 20th Chinese National Conference on Computational Linguistics*. Chinese Information Processing Society of China, 1218–1227.

[37] Francesco Logozzo. 2004. Automatic inference of class invariants. In *Intl. Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer.

[38] Ilya Loshchilov and Frank Hutter. 2017. Decoupled Weight Decay Regularization. (2017). https://doi.org/10.48550/ARXIV.1711.05101

[39] Jonathan I. Maletic and Michael L. Collard. 2015. Exploration, analysis, and manipulation of source code using srcML. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15)*. IEEE, 951–952.

[40] Bertrand Meyer. 1992. Applying 'Design by Contract'. *Computer* 25, 10 (Oct. 1992), 40–51. https://doi.org/10.1109/2.161279

[41] Bertrand Meyer. 1997. *Object-oriented software construction*. Vol. 2. Prentice hall Englewood Cliffs.

[42] Bertrand Meyer. 1998. Design by Contract: The Eiffel Method.. In *TOOLS (26)*. 446.

[43] Alysson F Milanez, Igor N S Ataıde, and Tiago L Massoni. 2021. Investigating the Use of JML Contracts. (2021).

[44] Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. 2022. Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code. (7 2022), 5546–5555.

[45] Robert O'Callahan and Daniel Jackson. 1997. Lackwit: A program understanding tool based on type inference. In *Proc. of the (19th) International Conference on Software Engineering*. IEEE Computer Society.
[46] OpenJML.org. 2024. *OpenJML*. https://www.openjml.org/
[47] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2001. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proc. of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02*. ACL, 311.
[48] David Lorge Parnas. 2011. Precise Documentation: The Key to Better Software. In *The Future of Software Engineering*, Sebastian Nanz (Ed.). Springer, 125–148.
[49] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can Large Language Models Reason about Program Invariants?. In *International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 27496–27520.
[50] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language Models Are Unsupervised Multitask Learners. *OpenAI blog* 1, 8 (2019), 9.
[51] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67.
[52] Todd W. Schiller, Kellen Donohue, Forrest Coward, and Michael D. Ernst. 2014. Case Studies and Tools for Contract Specifications. In *Proc. of the 36th International Conference on Software Engineering*. ACM, 596–607.
[53] Matthew Snover, Bonnie Dorr, Rich Schwartz, Linnea Micciulla, and John Makhoul. 2006. A Study of Translation Edit Rate with Targeted Human Annotation. In *Proc. of the 7th Conference of the Association for Machine Translation in the Americas: Technical Papers*. 223–231.
[54] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2019. MASS: Masked Sequence to Sequence Pre-Training for Language Generation. In *Proc. of the 36th International Conference on Machine Learning (Proc. of Machine Learning Research, Vol. 97)*. PMLR, 5926–5936.
[55] Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. 2019. Does BLEU score work for code migration?. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 165–176.
[56] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems*, Vol. 30. Curran Associates, Inc.
[57] Anh Duc Vu, Jan Arne Sparka, Ninon De Mecquenem, Timo Kehrer, Ulf Leser, and Lars Grunske. 2023. Contract-driven design of scientific data analysis workflows. In *2023 IEEE 19th International Conference on e-Science (e-Science)*. IEEE.
[58] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proc. of the 2023 Conference on Empirical Methods in Natural Language Processing*. ACL, 1069–1088.
[59] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proc. of the 2021 Conference on Empirical Methods in Natural Language Processing*. ACL, 8696–8708.
[60] Yi Wei, Carlo A Furia, Nikolay Kazmin, and Bertrand Meyer. 2011. Inferring better contracts. In *Proc. of the 33rd International Conference on Software Engineering*.
[61] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. 2012. *Experimentation in Software Engineering*. Springer.
[62] Haoze Wu, Clark Barrett, and Nina Narodytska. 2024. Lemur: Integrating Large Language Models in Automated Program Verification. In *The Twelfth International Conference on Learning Representations*.
[63] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. 2023. Evaluating Instruction-Tuned Large Language Models on Code Comprehension and Generation. (2023). https://doi.org/10.48550/ARXIV.2308.01240
[64] Jiyang Zhang, Marko Ristin, Phillip Schanely, Hans Wernher Van De Venn, and Milos Gligoric. 2022. Python-by-Contract Dataset. In *Proc. of the 30th ESEC/FSE*. ACM.

Received 2024-06-18; accepted 2024-08-15