TRATTO: A Neuro-Symbolic Approach to Deriving Axiomatic Test Oracles

DAVIDE MOLINELLI, Constructor Institute, Università della Svizzera italiana, Switzerland
ALBERTO MARTIN-LOPEZ, Università della Svizzera italiana, Switzerland
ELLIOTT ZACKRONE, University of Washington, USA
BEYZA EKEN, Università della Svizzera italiana, Switzerland
MICHAEL D. ERNST, University of Washington, USA
MAURO PEZZÈ, Constructor Institute, Università della Svizzera italiana, Switzerland

This paper presents TRATTO, a neuro-symbolic approach that generates assertions (boolean expressions) that can serve as axiomatic oracles, from source code and documentation. The symbolic module of TRATTO takes advantage of the grammar of the programming language, the unit under test, and the context of the unit (its class and available APIs) to restrict the search space of the tokens that can be successfully used to generate valid oracles. The neural module of TRATTO uses transformers fine-tuned for both deciding whether to output an oracle or not and selecting the next lexical token to incrementally build the oracle from the set of tokens returned by the symbolic module. Our experiments show that TRATTO outperforms the state-of-the-art axiomatic oracle generation approaches, with 73% accuracy, 72% precision, and 61% F1-score, largely higher than the best results of the symbolic and neural approaches considered in our study (61%, 62%, and 37%, respectively). TRATTO can generate three times more axiomatic oracles than current symbolic approaches, while generating 10 times less false positives than GPT4 complemented with few-shot learning and Chain-of-Thought prompting.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → **Neural networks**; *Natural language processing*.

Additional Key Words and Phrases: test oracle, automated oracle generation, deep learning, transfer learning

ACM Reference Format:
Davide Molinelli, Alberto Martin-Lopez, Elliott Zackrone, Beyza Eken, Michael D. Ernst, and Mauro Pezzè. 2025. TRATTO: A Neuro-Symbolic Approach to Deriving Axiomatic Test Oracles. 1, 1 (April 2025), 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Software testing is an expensive activity. While many testing activities can be automated, for example, test execution with JUnit [12], and unit test input generation with Randoop [41] and EvoSuite [26], other activities still require significant human effort. The oracle problem is still largely open and its automation is limited [4, 45]. Current approaches generate implicit and regression

Authors' Contact Information: Davide Molinelli, Constructor Institute, Università della Svizzera italiana, Switzerland, dm@constructor.org; Alberto Martin-Lopez, Università della Svizzera italiana, Switzerland, alberto.martin@usi.ch; Elliott Zackrone, University of Washington, USA, ezackr@cs.washington.edu; Beyza Eken, Università della Svizzera italiana, Switzerland, beyza.eken@usi.ch; Michael D. Ernst, University of Washington, USA, mernst@cs.washington.edu; Mauro Pezzè, Constructor Institute, Università della Svizzera italiana, Switzerland, mp@constructor.org.

ACM XXXX-XXXX/2025/4-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

oracles, which address only some aspects of the oracle problem: implicit oracles detect crashes and exceptions, while regression oracles detect differences in behavior between different versions.

In this paper, we focus on the automatic generation of *axiomatic oracles*. The current approaches to generate test oracles produce *concrete oracles*, that is, assertions on specific concrete inputs. For instance, `assertTrue(sum(20,22)==42)` indicates the expected result for the concrete input values 20 and 22. Axiomatic oracles generalize concrete oracles as assertions on the inputs and outputs variables and state. For instance `assertTrue(sum(a,b)==a+b)` predicates on the expected result for all pairs of inputs. Axiomatic oracles are essential elements in tools such as QuickCheck [14] and jmlunit [13] as an aid for testing. Axiomatic oracles are core assets of property-based testing [22], metamorphic relations [10], parameterized unit tests [53], and theories [48].

The benefits of axiomatic oracles are particularly significant when dealing with large, automatically generated test suites. Tools like Randoop [41] and EvoSuite [25] generate test cases that rely on implicit and regression oracles only, and may result in false alarms. Generating concrete oracles manually for each test input to filter out false alarms is extremely effort-demanding and impractical for large test suites. Axiomatic oracles in the form of preconditions, normal and exceptional postconditions within the unit under test are independent from the number of test cases, and can effectively reduce false alarms. Preconditions rule out invalid program inputs, thus reducing false positives. Normal and exceptional postconditions identify failures, thus reducing false negatives.

Beyond testing, axiomatic test oracles are useful for other software engineering (SE) tasks. For example, they aid program comprehension [16, 31] by specifying expected behavior unequivocally, unlike ambiguous natural language descriptions. They support requirements engineering [11] by providing formal specifications to validate and ensure completeness and consistency. In program synthesis [43], axiomatic oracles can guide the generation of correct code by offering clear specifications. Lastly, in runtime verification [33, 56], they serve as properties to monitor, ensuring the program meets its specifications during execution.

Current approaches for generating test oracles are either symbolic- or neural-based. Symbolic approaches derive axiomatic oracles, such as program invariants [20], metamorphic relations [8], and postconditions [6] from formal specifications [3, 18], software documentation [44], and program traces [57] using a pre-defined set of rules and patterns. Neural approaches leverage deep learning (DL) [19] and transfer learning [35] to generate either concrete test cases [54] or concrete oracles for a given test input [19, 59]. Symbolic and neural approaches both suffer from limitations. Neural approaches are well suited to handle fuzziness, for example, when deriving test oracles from ambiguous natural language descriptions. However, they require large amounts of (usually labeled) data and they are expensive to run. Symbolic techniques perform well by leveraging a fixed set of domain-specific rules, for instance, pattern and lexical matching [6–8], but do not generalize beyond the hard-coded rules. To the best of our knowledge, the potential of neural approaches for generating widely applicable axiomatic test oracles is still unexplored.

This paper introduces TRATTO, a neuro-symbolic approach to derive axiomatic oracles from commonly available software artifacts such as source code and documentation. An axiomatic oracle is composed of lexical tokens.[1] For instance, the axiomatic oracle "`result>0;`" is composed of four tokens, '`result`', '`>`', '`0`', and '`;`'. TRATTO (TRAnsformer-based Token-by-Token Oracle generation) reformulates the oracle generation problem as a token generation problem.

TRATTO generates oracles token by token. At each token-generation iteration, TRATTO restricts the search space of the next possible tokens in two ways. The first restriction is based on a

[1]Unless otherwise specified, we use "token" in the traditional sense of compilers, lexers, and programming languages. Machine learning transformers use "token" for a different concept (e.g., the *lexical* token `someVar` may be *tokenized* into two *transformer* tokens, `some` and `Var`).

programming language grammar and the portion of the oracle generated so far. For instance, a boolean expression cannot be the argument with a “>” operator. The second restriction is based on the symbols that are in scope. These symbols include the method parameters and the return value (for postconditions), fields and methods both in the current class and accessible through them.

TRATTO implements a neural approach to select a token from the set of available tokens. TRATTO leverages pre-trained transformers fine-tuned on the task of selecting candidate tokens based on (i) the oracle generated so far, (ii) the unit under test (including source code and documentation), and (iii) additional unit context, for instance, information about the available APIs.

The neural module of TRATTO features a multitask model trained on a dataset that was expressed in two different ways (for two different tasks): a dataset of oracles, and a dataset of tokens that appear in the oracles. The dataset substantially extends the dataset available in the replication package of Blasi et al. [1, 6]. We enhanced the initial dataset by (i) fixing semantically-incorrect oracles, (ii) adding oracles that we could infer from the code and the documentation, (iii) adding oracles from other publicly available Java projects, and (iv) automatically augmenting the Javadoc comments with semantically equivalent Javadoc comments that refer to the same oracles. The resulting dataset features 34,249 oracle samples. We produced a dataset of 188,900 tokens from the tokens in the oracles. Both datasets are publicly available in multiple formats, suitable for training sequence-to-sequence and classification models, to support future research [2].

We experimentally evaluated TRATTO. We compared the performance of different code models to support the neural module, Code Gemma (Google) [51], StarCoder2 (BigCode) [34] and Code Llama (Meta) [47], all in their version featuring 7B trainable parameters. We selected Code Llama since it outperforms the other models for this task, with 91% accuracy in predicting the next oracle token.

We performed two ablation studies to compare TRATTO with (i) a purely neural model trained for predicting oracles, not supported by the token-by-token symbolic approach, and (ii) a version of TRATTO featuring two different models, instead of a single multitask model. The results show that the symbolic module and the multitask model provide an additional +6 and +3 percentage points of accuracy in predicting oracles, respectively.

We compared TRATTO with two state-of-the-art symbolic and neural approaches for axiomatic oracle generation, Jdoctor (symbolic) [6] and GPT4 (neural) [39] on a ground-truth dataset of axiomatic test oracles. The experimental results that we discuss in Section 4.3.5 indicate that TRATTO outperforms both Jdoctor and GPT4 in terms of *accuracy* (73% for TRATTO, 61% for Jdoctor, 40% for GPT4), *precision* (72% for TRATTO, 62% for Jdoctor, 24% for GPT4), and *F1-score* (61% for TRATTO, 25% for Jdoctor, 37% for GPT4). TRATTO’s *recall* (52%) is better than Jdoctor’s (16%) and worse than GPT4’s (89%). In overall terms, TRATTO achieves an excellent balance between generating oracles (3× more than Jdoctor) while incurring in few false positives (10× less than GPT4).

We also compared the robustness of TRATTO, Jdoctor and GPT4 in generating correct oracles when the documentation of the units under test was modified in different ways (details in Section 4.4). The results show that TRATTO and GPT4 have comparable performance, generating 195 and 208 correct and compilable oracles, respectively, from a set of 220 descriptions derived from 55 original descriptions. Jdoctor generated only 49 correct and compilable oracles.

Lastly, we evaluated the effectiveness of the oracles generated by TRATTO and Jdoctor for increasing the mutation score of test suites automatically generated with EvoSuite [25]. We could not evaluate GPT4 for this purpose due to the large amount of non-compilable oracles generated (552 out of 1,213), which made it impossible to automate the insertion of oracles into the test cases. We considered two types of test suites: with implicit oracles only, and with both implicit and regression oracles. From among 6 projects considered, TRATTO increased the mutation score of 5 test suites with implicit oracles and 3 test suites with both implicit and regression oracles, while Jdoctor increased the mutation score of 4 test suites in the former case and in no cases in the latter.

In summary, this paper discusses the limitations of the state-of-the-art approaches to generate axiomatic oracles (Section 2), and makes the following contributions:

(1) It defines a novel approach that iteratively generates test oracles, token by token, by combining a symbolic with a neural approach to steer the generation of tokens toward valid oracles, thus reducing the impact of false positives of purely neural approaches (Section 3.1).
(2) It introduces Tratto, a neuro-symbolic approach and corresponding tool to generate axiomatic oracles from source code and documentation (Sections 3.2 and 3.3).
(3) It proposes a collection of comprehensive datasets of oracles and tokens that can be reused for training future models for generating oracles (Section 3.4).
(4) It performs empirical studies to: (i) compare the performance of different code models for the tasks of oracle evaluation and token selection; (ii) highlight the contributions of Tratto's components to its overall performance; and (iii) compare Tratto with state-of-the-art approaches for oracle generation, showing its superior performance in terms of correctness, robustness and applicability to enhancing automatically generated test suites (Section 4).

2 Motivating Example

This section shows how state-of-the-art oracle generators perform on an example method (Listing 1). Their limitations motivate Tratto.

The documentation in Listing 1 describes a precondition "`series >= 0`" in natural language (*"the series index (zero based)"*, line 6). If a test generator produces a unit test case with a negative `series` as argument, the test crashes. The precondition indicates that the test case is invalid, and avoids executing the test with a false positive result.

```
 1 /**
 2  * Sets the item label generator for a series and
 3  * sends a {@link RendererChangeEvent} to all
 4  * registered listeners.
 5  *
 6  * @param series the series index (zero based).
 7  * @param generator the generator
 8  * (<code>null</code> permitted).
 9  *
10  * @see #getSeriesItemLabelGenerator(int)
11  */
12 public void setSeriesItemLabelGenerator(int series,
13         CategoryItemLabelGenerator generator) {
14     setSeriesItemLabelGenerator(series, generator,
15             true);
16 }
```

Listing 1. Documentation and implementation of method setSeriesItemLabelGenerator from JFreeChart.

The main state-of-the-art oracle generators do not generate this precondition:
TOGA [19], a neural approach for generating test assertions, does not generate preconditions at all.
Jdoctor [6], a symbolic approach, does not generate the precondition either, since it is based on a set of patterns and rules that do not match the comment.
EvoSuite [25], a search-based test generator, generates regression test oracles, and no preconditions.
GPT4 [40], a large language model (LLM) trained on natural language and code, correctly generates the precondition, but also generates wrong, useless, or non-compilable oracles such as "`generator == null || generator != null`" and "`generator == null || generator is a valid CategoryItemLabelGenerator instance`".
Tratto derives the precondition "`series >= 0`" that prevents the generation of invalid test cases and false positives, and does not generate wrong oracles like GPT4. The neural module of Tratto generalizes to previously unseen oracles, beyond the fixed set of rules and patterns of Jdoctor, while the symbolic module restricts the possible oracles to compilable oracles.

3 Tratto

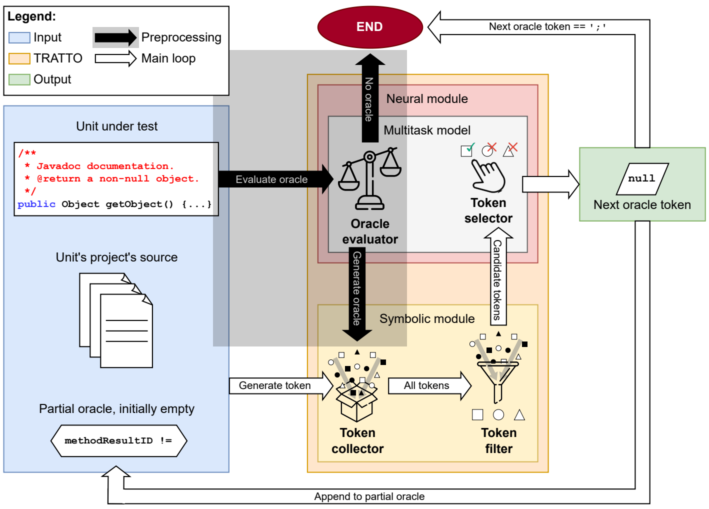
Tratto generates executable preconditions, regular and exceptional postconditions that together comprise axiomatic oracles. Tratto generates executable assertions in Java from Java source code and Javadoc comments. Preconditions, like *"param1 cannot be null"*, constrain the validity of the test input. Postconditions, like *"the output must be positive"*, and exceptional postconditions, like *"if param1 is null, an exception is thrown"*, constrain the expected behavior of a program.

The input of TRATTO is the source code of the method under test and its context, such as fields and methods in its class. TRATTO tries to generate a precondition from each `@param` tag, a regular postcondition from each `@return` tag, an exceptional postcondition from each `@throws` and `@exception` tag, and preconditions, regular and exceptional postconditions from the free-text part of the Javadoc, the method signature and the implementation.

3.1 Architecture

TRATTO integrates a symbolic and a neural module to generate test oracles iteratively, token by token. The symbolic module features a *Token collector* and a *Token filter* to restrict the possible tokens forming the oracle. The neural module consists of a multitask model that works in two modes: *Oracle evaluator* and *Token selector*. The former decides whether to generate an oracle or not, while the latter selects the tokens that incrementally form an oracle, based on those returned by the symbolic module.

Figure 1 shows the workflow of TRATTO. As a preprocessing step (black arrows), the *Oracle evaluator* component of the neural module decides if an oracle should be generated, solely based on the documentation and code of the *unit under test*. If so, it triggers the token-generation workflow (white arrows). At each iteration, TRATTO generates the next token of the oracle based on (i) the unit under test, (ii) the *unit project's source* (documentation and source of the whole project), and (iii) the *partial oracle* (portion of the oracle built so far). TRATTO starts with an empty partial oracle and terminates when the next generated token is a semicolon (';').



```
/**
 * Javadoc documentation.
 * @return a non-null object.
 */
public Object getObject() {...}
```

```
methodResultID !=
```

```
null
```

Fig. 1. Workflow of TRATTO.

The symbolic and the neural modules of TRATTO cooperate to incrementally generate the tokens that comprise the partial oracle, until they produce a complete oracle. At each iteration, the *Token collector* retrieves all possible tokens to build an oracle, for instance, parameters of the unit under test with their fields and methods. Then, the *Token filter* selects the subset of tokens that are *valid*, i.e., that can be added to the partial oracle to eventually generate a *well-formed* assertion (*Candidate tokens* in Figure 1). For example, the `instanceof` operator may not follow a variable of primitive type, although it is a syntactically valid expression. Given a set of candidate tokens, the *Token selector* of TRATTO's neural module selects the next token that shall be added to the partial oracle.

TRATTO uses a novel token-level approach to generate oracles. The state-of-the-art approaches like TOGA [19] and Jdoctor [6] generate a whole oracle at once. Generating test oracles token-by-token brings two main benefits:

(1) The grammar-based symbolic approach of Tratto: (i) generates only oracles that are compilable by construction, while neural-based approaches may generate oracles that do not compile [40, 54]; (ii) generates any oracle that is consistent with the grammar, while some neural approaches generate only oracles compatible with a set of candidate assertions [19]; and (iii) collects contextual information from the unit under test useful to generate an oracle, for instance, possible method calls to external libraries, calls that neither symbolic nor neural approaches can access without manual intervention, while prompt-based neural approaches require manually specified data. We define the grammar-based symbolic approach of Tratto in Section 3.2.

(2) The neural model: (i) is trained on a dataset of tokens much larger than a dataset of oracles, since oracles are composed of multiple tokens; and (ii) takes full advantage of transformers capabilities of learning patterns and associations related to natural language and source code, by relying on training with significantly more data, beyond the limitations of semantic approaches that rely on pattern, lexical and semantic matching. We discuss the neural module of Tratto in Section 3.3 and the benefits of the tokens dataset in Section 3.4.

3.2 Symbolic Module

The symbolic module of Tratto restricts the search space of the possible oracles for a given unit, to a set of tokens that produce compilable oracles, by construction. The *Token collector* collects all possible tokens that could be used to form an oracle. The *Token filter* discards the illegal tokens (syntactically and semantically invalid tokens that would make the oracle non-compilable) with a grammar-based approach.

3.2.1 Token Collector. The *Token collector* creates a set of all the tokens that Tratto needs to generate oracles for the target unit. These tokens may be *generic* tokens related to the target unit under test, collected in the first token-generation iteration, or *specific* tokens related to the current partial oracle, collected only in some iterations.

In the first iteration (when the partial oracle is empty), the *Token collector* gathers three types of *generic* tokens: (i) common tokens which could be part of any oracle, such as operators, keywords, and common constants like 0 and 1; (ii) tokens extracted from the project under test, including classes and their respective fields and methods (e.g., `CollectionUtils.isEmpty()`); and (iii) tokens extracted from the method under test, including its parameters (if any) as well as fields and methods callable upon those parameters, containing class, and return value of the method (e.g., `this.`**`contains`**`(o)`).

At each subsequent iteration, the *Token collector* may augment the set of tokens with *specific* tokens that may occur when the last token of the partial oracle is a period that follows an expression that is a class of the project, `this`, `methodResultID` (which identifies the return value of the method under test), or a method parameter. For instance the *Token collector* adds the tokens '`hasNext`' and '`next`' to the set of tokens when the partial oracle ends with "`this.iterator().`".

3.2.2 Token Filter. The *Token filter* discards both the tokens that are syntactically illegal in the next position of the partial oracle (for instance, '`true`' is not syntactically correct as the next token after "`arg1 >`"), and the tokens that would result in a compilation error of the oracle (for instance, the > operator may follow only an expression that evaluates to a numeric type). The *Token filter* prunes the set of tokens with a grammar-based approach. Tratto also implements several *context restrictions* that discard tokens that may not occur in the next position of the oracle, even if grammatically legal, for instance, `methodResultID` may not be used if the method under test is void.

Tratto Grammar. Axiomatic oracles are boolean expressions. Tratto implements a custom-defined grammar to express oracles (available in the replication package [2] due to space restrictions). In a nutshell, the grammar supports expressions consisting of variables potentially chained with class fields and method calls, such as "`this.a.b()`"; arithmetic operations, as in "`arg2+arg3`"; comparison operators, as in "`methodResultID != null`"; and conjunctions and disjunctions formed with the logical operators `&&` and `||`. This simple but effective grammar can represent all procedure specifications from the work by Blasi et al. [6], while permitting new oracles.

Context Restrictions. A grammar does not suffice to generate a test oracle, since it lacks the underlying semantics of the specific tokens conforming it. For example, according to the Tratto grammar, the token '`instanceof`' is valid after the partial oracle "`methodResultID`". However, if the type of the variable `methodResultID` (i.e., the return type of the method under test) is primitive, the resulting expression would not compile. We refer to cases like this as *context restrictions*, since they restrict tokens depending on the context of the oracle (so far). Tratto implements 27 context restrictions (documented in the replication package [2]). The context restrictions also alleviate the load of the neural module, by reducing the number of tokens that it must evaluate to generate accurate oracles.

3.3 Neural Module

The neural module comprises the *Oracle evaluator*—decides whether an oracle should be generated or not—and the *Token selector*—guides the generation of oracles toward optimal solutions, token by token, in all iterations. The neural module features a multitask DL model trained for both tasks.

3.3.1 Oracle Evaluator. Although the oracle evaluation task is a binary task that determines whether an oracle should be generated or not, we treat it as a generation task, so that we can reuse the same auto-regressive model (e.g., specialized for code generation and completion) for both tasks. We ask the model to fill out the mask of a masked input. Listing 2 shows the input template used for this task. It is composed of an oracle type and a Javadoc tag, if available (line 1), two next possible candidate tokens for the partial oracle (line 2), the mask token to fill out (line 4), and the method under test (lines 6–8). The model selects one of the two next possible tokens, either '`assertTrue(`' (an oracle should be generated) or '`// No assertion possible`' (no oracle).

3.3.2 Token Selector. We treat the token selection task as a generation task, as for the *Oracle evaluator*. Listing 3 shows the input template that we use for this task. With respect to the input of the *Oracle evaluator*, this is augmented with (i) a list of next possible tokens and not just a binary choice (line 2), (ii) the partial oracle that is not empty (line 4), and (iii) the additional context with information about method signatures and field declarations corresponding to the next possible tokens (lines 10-11), aiming to provide more context to the model. The model *selects* one of the next possible tokens that produces a new correct partial oracle when added to the tail of the input partial oracle.

```
1 // <oracle_type>: "<javadoc_tag>"
2 // Next possible tokens: ['assertTrue(',
    '// No assertion possible']
3 // Assertion:
4 <FILL_ME>
5
6 // Method under test:
7 <method_javadoc>
8 <method_source>
```

Listing 2. Input template for the Oracle Evaluator.

```
1  // <oracle_type>: "<javadoc_tag>"
2  // Next possible tokens: [<next_possible_tokens>]
3  // Assertion:
4  assertTrue(<partial_oracle><FILL_ME>
5
6  // Method under test:
7  <method_javadoc>
8  <method_source>
9
10 // Additional context:
11 <method_signatures_and_field_declarations>
```

Listing 3. Input template for the Token Selector.

Listing 4 is a complete example of the input to the model when acting as *Token selector*. The example is a snapshot of the construction of an exceptional postcondition as shown in line 1 (partial oracle = "`array.getClass().`" in line 4). As the `getClass()` method returns an object of type `Class`, the next possible tokens are non-private fields and methods of class `Class` (line 2). The model selects the correct token, in this case '`isArray`' (additional information provided in line 24), among the possible tokens, to replace the mask token in the partial oracle, thus continuing the oracle generation process.

```
1  // Exceptional postcondition: "@throws
     IllegalArgumentException if <code>array</code> is
     not an array."
2  // Next possible tokens: ['equals', 'toString',
     'isArray', 'getClassData', 'getClassLoader', ...]
3  // Assertion:
4  assertTrue(array.getClass().<FILL_ME>
5
6  // Method under test:
7  /**
8    * Constructs an ArrayListIterator that will
9    * iterate over the values in the specified array.
10   *
11   * @param array the array to iterate over
12   * @throws IllegalArgumentException if
13   * <code>array</code> is not an array
14   * @throws NullPointerException if
15   * <code>array</code> is <code>null</code>
16   */
17 public ArrayListIterator(final Object array){
18     super(array);
19 }
20
21 // Additional context:
22 public boolean equals(Object arg0)
23 public String toString()
24 public native boolean isArray()
25 Object getClassData()
26 public ClassLoader getClassLoader()
27 ...
```

Listing 4. Input to the model acting as Token Selector.

3.4 Data Collection

We trained Tratto's neural model used for the *Oracle evaluator* and *Token selector* with datasets of oracles and tokens, respectively. Figure 2 shows how we generated the datasets. We started with an existing dataset of procedure specifications [6], added comments with and without a corresponding oracle, and manually inspected and cleaned the dataset. We augmented this dataset via automated techniques. We disaggregated oracles into tokens via the *Token collector* and *Token filter* components of Tratto, to produce the tokens dataset. Oracles and tokens datasets are available in the replication package [2].

3.4.1 Procedure Specifications Dataset. Blasi et al. [6] provide a dataset of *procedure specifications* (preconditions, regular and exceptional postconditions) that correspond to Javadoc tags. The procedure specifications express the intended behavior of a program and therefore they can be used as test oracles. This dataset contains 3,150 tuples in the form of $\langle (u, jt), o \rangle$, where u is the *unit* under test, jt is the *Javadoc tag* in the unit documentation, and o is an executable Boolean expression that corresponds to the Javadoc tag and may be used as an *oracle*. Blasi et al. [6] provide also 23,397 Javadoc tags for which their approach cannot generate axiomatic oracles.

3.4.2 Oracles Dataset. We created an initial oracles dataset by combining the 3,150 procedure specifications from the dataset of Blasi et al. [6] (*positive instances*) with 3,150 randomly sampled *negative instances* (*Comments without oracles* in Figure 2) from among the 23,397 Javadoc tags without oracles. Since the authors report an average precision and recall of 92% and 83%, respectively, we decide to inspect the selected 6,300 instances (3,150 positive and negative) to fix possible mistakes (e.g., wrongly generated oracles). Additionally, we add 222 oracles from various Java projects on GitHub to increase the quality of the final dataset. This manual process (step ① in Figure 2) leads to an initial oracles dataset of 5,911 samples, 4,582 of which are positive and 1,329 of which are negative. Indeed, a large number of the 3,150 inspected comments without oracles resulted in oracles that could be derived, hence the larger number of positive instances in the resulting dataset.

We augmented the oracles dataset by creating semantically equivalent versions of the comments in the dataset with ChatGPT [38]. We asked it to generate new Javadoc comments by suggesting equivalent versions of

```
@return the sum {@code a + b}. // Original
@return the total value of {@code a + b}
@return the result of adding {@code a} and {@code b}
@return the outcome of summing {@code a} and {@code b}.
@return the value obtained by adding {@code a} and {@code b}.
@return the sum of {@code a} and {@code b}
```

Listing 5. Equivalent Javadoc tags generated with ChatGPT.

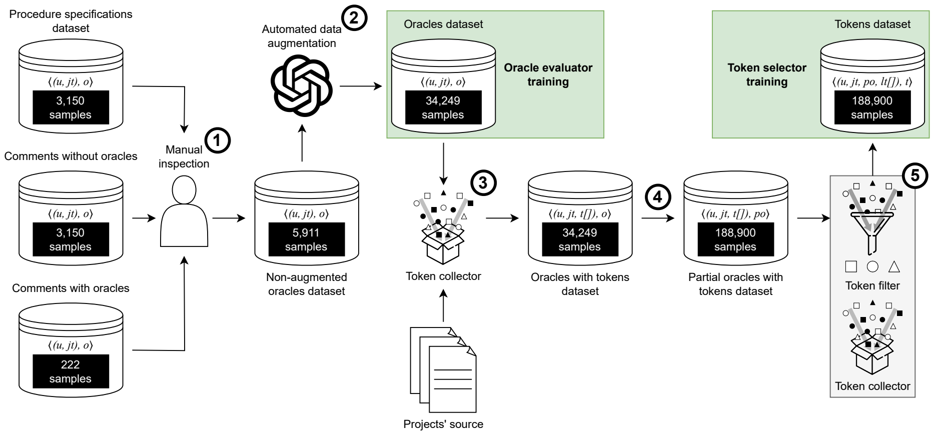


Fig. 2. Data collection process.

the comments, with a maximum of 5 (②). Listing 5 shows a sample set of equivalent Javadoc tags generated with ChatGPT. Details of the prompts and data cleaning are available in the replication package [2]. We obtained 34,249 comments, 23,392 with an oracle (positive samples) and 10,857 without (negative samples).

3.4.3 Tokens Dataset. Based on the oracles dataset, we generated an *oracles with tokens dataset*, by leveraging the *Token collector* to extract *generic* tokens that could belong to each oracle (③), for instance, class names (e.g., `HashBiMap`), constants (e.g., `CollectionUtils.EMPTY_COLLECTION`), and method names (e.g., `.toString()`). The *Token collector* component (Section 3.2.1) analyzes the source code of the project related to the oracle to extract available tokens, and produces a dataset of tuples in the form of ⟨*(u, jt, t[]), o*⟩, where *t[]* refers to the *list of all possible tokens* that could be initially used to start building the oracle *o*.

Subsequently, we automatically disaggregated oracles into partial oracles (④) and used both the *Token collector* and *Token filter* to add relevant tokens and rule out invalid tokens for each partial oracle (⑤), respectively, leading to a dataset of tokens in the form of $\langle(u, jt, po, lt[]), t\rangle$ tuples, where *po* denotes a *partial oracle*, *lt[]* refers to the *list of legal tokens* that could possibly follow the partial oracle (e.g., '`0`', '`1`', and '`SomeClass`' could follow "`result >`"), and *t* is the actual next token after the partial oracle, which must be one of the tokens from the aforementioned list. The tokens dataset contains 188,900 samples in total.

Figure 3 shows an example of the conversion process from an oracle sample to token samples. The oracle "`loadFactor<=0;`" represents an exceptional behavior, i.e., if the argument of the method under test `loadFactor` is less than or equal to 0, an exception should be thrown. This oracle contains four tokens, so the data generation process produces four partial oracles. For instance, in the first iteration, the oracle may start with tokens such as parameters from the method under test (i.e., `loadFactor` and `initialCapacity`), an opening parenthesis, the name of a class (e.g., if a static method is

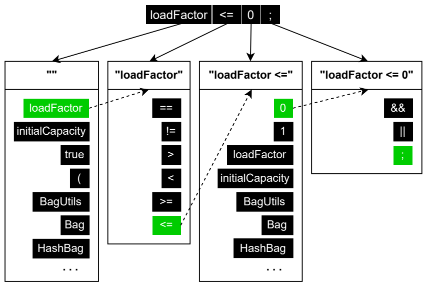


Fig. 3. Converting one oracle sample into four token samples. Oracle (o) at the top; partial oracles (po) and legal tokens ($lt[]$) on top and bottom white boxes, respectively; next tokens (t) in green.

to be called), and so on. In the second iteration, when the partial oracle is "`loadFactor`", the next token can only be a comparison operator since `loadFactor` is of numeric type, thus the *Token filter* discarded all other tokens that may not be used in this context (including arithmetic operators which are not allowed in this particular position according to the Tratto grammar [2]). The same process is repeated for the remaining tokens of the oracle.

4 Evaluation

Our experimental evaluation addresses the following research questions:

RQ$_1$: *How do different LLM code models compare for evaluating oracles and selecting tokens?* We evaluate different code models aiming to find the most suitable one for the tasks performed by the neural module of Tratto.

RQ$_2$: *What is the contribution of the symbolic module and the multitask model to the overall performance of Tratto?* We carry out ablation studies to assess the performance of Tratto without the symbolic module (a purely neural model not supported by the proposed token-by-token oracle generation approach) and without the multitask model (two different models for the *Oracle evaluator* and *Token selector*).

RQ$_3$: *What is the effectiveness of Tratto in generating axiomatic test oracles and how does it compare with state-of-the-art neural and symbolic techniques?* We devise a ground-truth dataset of axiomatic test oracles to evaluate Tratto, and compare it against state-of-the-art symbolic- and neural-based approaches for oracle generation.

RQ$_4$: *How robust is Tratto to documentation variations?* Based on the results of RQ$_3$, we select the oracles that all approaches correctly predict, we apply systematic variations to their associated documentation, and we measure the robustness of Tratto and the other approaches in terms of the amount of oracles that they can still correctly infer.

RQ$_5$: *How effective are the generated oracles for improving test suites?* We apply the oracles that Tratto generates to test suites automatically generated with EvoSuite, and we measure the effectiveness of Tratto for testing as the improvement of the mutation score.

4.1 RQ$_1$: Code Models Comparison

RQ$_1$ addresses the suitability of LLM code models that Tratto uses to both *evaluate oracles* and *predict the next oracle token*. We compare three models—Code Gemma by Google [51], StarCoder2 by BigCode [34], and Code Llama by Meta [47]—to determine the most effective LLM for such tasks. We model both tasks as code infilling problems (see Listings 2 and 3). We select these models since they are state-of-the-art, support infilling problems, are trained on large-scale code corpora, and are popular among researchers and practitioners for addressing code-related tasks. We used the version of each model that features 7B trainable parameters.

The dataset used for training and validation features 223,149 samples (34,249 oracles + 188,900 tokens). We set aside all samples belonging to a single Java project (Guava, 30,891 samples, 14% of the total) for validation, and used the remaining 192,258 samples (86%) for training.[2] We trained all models for two epochs, as we experimentally determined that accuracy does not improve significantly after this point. We set the input length to 2,048 transformer tokens and the output length to 32 transformer tokens.[3] We used default hyper-parameters for all trainings.

The accuracy obtained with Code Gemma, StarCoder2, and Code Llama is 58%, 69%, and 91%, respectively. Code Llama largely outperforms both Code Gemma and StarCoder2. We hypothesize

[2]We split the dataset in this way to avoid data leakage, since some oracle and token samples from the same project are the same except for the rephrased Javadoc tag, according to the data augmentation process described in Section 3.4.2.
[3]Although the output of the models is a single *lexical* token (e.g., a variable name such as `millis2secs`) this is tokenized into multiple *transformer* tokens (e.g., `mill-is-2-se-cs`, five tokens).

that such a difference is due to both the pre-training data and the specificity of each training procedure. Code Llama is pre-trained primarily on natural language and actual code, and is further specialized for long-context inputs, which fits well with our tasks. StarCoder2 is pre-trained on varied data, including GitHub issues, pull requests and Jupyter notebooks. Code Gemma is pre-trained on English language data from open source mathematics datasets and synthetically generated code. Both Code Gemma and StarCoder2 are pre-trained on general data and synthetic code that are not particularly useful for our case.

The results of RQ_1 measure the accuracy of different LLM code models in correctly evaluating oracles and selecting oracle tokens, and clearly point to Code Llama as the best choice to generate correct axiomatic oracles. When used to generate a complete sequence of tokens conforming an oracle, TRATTO instantiated with Code Llama correctly generates 69% axiomatic oracles on the Guava validation set, that is, 3,337 oracles (out of 4,865) have all their tokens correctly predicted.[4]

Answer to RQ_1: Code Llama achieves 91% accuracy for the tasks of evaluating oracles and selecting tokens, and outperforms both Code Gemma (58%) and StarCoder2 (69%).

4.2 RQ_2: Ablation Studies

RQ_2 addresses the contributions of the symbolic module and multitask model of TRATTO in generating oracles. To this end, we carry out two ablation studies where these components are removed.

In the first study, we remove the token-by-token oracle generation approach enabled by the symbolic module. Thus, the neural module either infers the whole oracle or indicates the impossibility of generating one, solely based on the unit under test. We set aside the Guava oracles (4,865) for validation, and used the remaining (29,384) for fine-tuning the Code Llama 7B model with default hyper-parameters for two epochs, as we did for TRATTO. The resulting model generates 3,049 correct oracles out of 4,865, achieving an accuracy of 63%, 6% less that the 69% accuracy of TRATTO. The difference is statistically significant according to the McNemar test (p-value < 0.001, Odds Ratio = 2.57), which is suitable to pairwise compare dichotomous results of two different treatments as in this case [36]. This indicates that the symbolic module and the token-by-token approach contribute to improve the performance of generating axiomatic oracles.

In the second study, we remove the multitask model and fine-tune two separate models, an *Oracle evaluator* and a *Token selector*. Both are based on Code Llama 7B and the training is performed with default hyper-parameters for two epochs. The *Oracle evaluator* is fine-tuned on the 29,384 oracle samples (86% of all oracles) that do not include the Guava project, and the *Token selector* is fine-tuned on the respective 162,874 token samples (86% of all tokens). Then, we jointly evaluate the capabilities of both models to correctly predict the 4,865 oracles from the Guava validation set, i.e., with the *Oracle evaluator* correctly predicting whether an oracle should be generated and, if so, with the *Token selector* correctly predicting all tokens of the oracle. Overall, this approach achieved 66% accuracy in generating oracles (3,213 out of 4,865 oracles from the Guava validation set), 3% less than the accuracy of TRATTO. The difference is statistically significant (p-value < 0.001, Odds Ratio = 1.55). This highlights the benefits of leveraging a multitask model for evaluating oracles and predicting tokens.

Our analysis of the oracles that TRATTO generates and the ablated approaches fail to generate indicates that the ablated approaches either incur in false positives, i.e., they generate oracles where there should be none, or they generate incorrect oracles. As an example of false positives, the purely neural model wrongly generates the precondition “`array != null`” from the Javadoc

[4]Note that this dataset also contains non-oracle samples (*Comments without oracles* in Figure 2), which are deemed as *correctly predicted* if the *Oracle evaluator* judged no need to generate an oracle.

tag *"@param array an array of {@code short} values, possibly empty"*. The *Oracle evaluator* of the non-multitask model generates a precondition from the Javadoc tag *"@param defaultValue the value provided for inputs absent in map keys"*, which does not express any precondition. As an example of wrong oracles, the purely neural model generates the incorrect oracle "`true ? Arrays.stream(array).anyMatch(jdVar -> jdVar == target) : true`", which asserts that the `array` parameter should always contain the `target` parameter, from the Javadoc tag *"@return {@code true} when any element {@code array[i]} equals {@code target}"*, while TRATTO generates the correct oracle, "`Arrays.stream(array).anyMatch(jdVar -> jdVar == target) ? methodResultID == true : methodResultID == false`", which asserts that, if the `array` parameter contains the `target` parameter, the method should return `true`, otherwise it should return `false`.

Answer to RQ$_2$: The purely neural and the non-multitask approaches achieve 63% and 66% accuracy, respectively, less than the 69% accuracy of TRATTO. Thus, we conclude that the symbolic module and the multitask model both improve the performance of TRATTO.

4.3 RQ$_3$: Oracle Generation

RQ$_3$ compares the effectiveness of TRATTO to the state-of-the-art neural and symbolic approaches for generating axiomatic oracles. We present the ground-truth dataset used for evaluating, the state-of-the-art approaches we compare TRATTO with, the metrics we computed with the experiments, the training and evaluation setup of the experiments, and finally we discuss the results.

4.3.1 Ground-Truth Dataset. We manually created a ground-truth dataset of axiomatic test oracles, to fairly compare TRATTO with state-of-the-art approaches. We extracted and validated oracles from Defects4J [30]. We excluded two projects (Apache Commons Math [24] and Apache Commons Collections [23]) that are part of the training set of TRATTO. For each of the remaining 15 projects in Defects4J, we systematically selected 10 Java classes by sorting them according to their character count and selecting those evenly spaced within the 5% to 95% range (i.e., the classes at the percentiles 5, 15, 25, etc.). We manually extracted all possible axiomatic oracles from all methods of each selected class. The dataset contains 389 axiomatic oracles (*positive samples*) belonging to 274 methods of 150 classes from 15 projects. For each method, we also generate one *negative sample* if no precondition, postcondition or exceptional postcondition could be generated for such method. For instance, if a method encodes two preconditions, it will result in two positive samples (two preconditions) plus two negative samples (one *non-postcondition* and one *non-exceptional-postcondition*). This leads to a total of 496 negative samples. Each sample was reviewed by at least two authors.

4.3.2 Comparison Approaches. We compare TRATTO with Jdoctor [6] and GPT4 [40], as representative symbolic and neural approaches, respectively. Jdoctor [6] generates axiomatic oracles (preconditions, normal and exceptional postconditions) from the Javadoc tags `@param`, `@return` and `@throws`/`@exception`, respectively, by applying pattern, lexical, and semantic matching techniques. Jdoctor outperforms other approaches such as Toradocu [27] and @tComment [50].

The state-of-the-art neural approaches to generate oracles [19, 55, 59] produce concrete test oracles, that is, assertions on specific concrete inputs. They are not directly comparable with TRATTO, which generates axiomatic oracles that predicate on variables, and are valid for all concrete inputs. Thus we compare TRATTO against GPT4 (model GPT4-o) as representative neural approach to generate axiomatic oracles. We enhance GPT4 with few shot learning [58] combined with Chain-of-Thought prompting [60] to fully leverage its understanding capabilities. We provide GPT4 with three examples of how and when to generate axiomatic oracles based on several Java methods, following a step-by-step approach, and ask it to generate oracles for the ground-truth dataset, method by method. We share the prompts we use in our experiments in the replication package [2].

4.3.3 Metrics. We comparatively evaluate TRATTO by computing *true positives* (correctly predicted oracles), *true negatives* (instances for which no oracle should be generated, and none is generated), *false positives* (instances for which no oracle should be generated, but one is generated, or wrongly generated oracles), and *false negatives* (oracles not generated, while one should be generated).

We classify incomplete oracles, that is, oracles that capture only a subset of the correct behavior, as *false positives*, since they partially miss the semantic of the reference oracle and do not work for all test cases. The oracle "`result != null`" is an example of *incomplete* oracle for a method that returns a positive `Integer`, as it correctly captures only a subset of the expected behavior, while "`result >= 0`" is a *wrong* oracle for the same method, as `0` is not positive.

We also compute *accuracy* (portion of all correct predictions out of the total predictions), *precision* (portion of oracles correctly generated out of all oracles generated), *recall* (portion of oracles correctly generated out of all actual oracles in the dataset), and *F1-score* (weighted harmonic mean of precision and recall). *Accuracy* measures the overall performance of the technique, but does not differentiate between *false positives* and *negatives*. *Precision* measures the ability to avoid false positives. *Recall* reflects the ability to capture all relevant instances, avoiding false negatives. *F1-score* combines *precision* and *recall* into a single measure that balances false positives and negatives, and is especially useful when the dataset is unbalanced and the positive class is rare, as in our case.

4.3.4 Training and Evaluation Setup. We trained TRATTO with the same setup as in previous experiments, although using the complete dataset (223k samples) for training. We generated as many axiomatic oracles as possible for all methods in the ground-truth dataset. We downloaded and set up the most recent version of Jdoctor from the publicly available GitHub repository (commit d76899f). Jdoctor does not need training, as it is based on a set of heuristic rules and patterns. We generated as many axiomatic oracles as possible for all methods in the ground-truth dataset, as we did for TRATTO. We performed similarly for GPT4, generating as many axiomatic oracles as possible for all methods, by crafting a prompt per method, making an API call per prompt, and collecting all responses, which we manually analyzed.

4.3.5 Results. Table 1 reports the *accuracy* (row A), *precision* (P), *recall* (R), and *F1-score* ($F1$) that we computed for each approach (TRATTO, Jdoctor and GPT4) and each project (columns *closure-compiler ... mockito*), as well as for all projects (column *Total*). The table highlights in green the best *Total* values and in red the worst values. We observe that TRATTO outperforms both Jdoctor and GPT4 in terms of *accuracy* (73% for TRATTO, 61% for Jdoctor, 40% for GPT4), *precision* (72% for TRATTO, 62% for Jdoctor, 24% for GPT4), and *F1-score* (61% for TRATTO, 25% for Jdoctor, 37% for GPT4). TRATTO's *recall* (52%) is better than Jdoctor's (16%) and worse than GPT4 (89%). The results indicate that TRATTO infers more correct predictions than both Jdoctor and GPT4 (*accuracy*) with a low impact of wrong results (*precision*). The better performance of GPT4 than TRATTO in terms of *recall* indicates that GPT4 generates a higher proportion of oracles than TRATTO out of the ground truth. This was expected since GPT4 is a general-purpose LLM with better generalization capabilities. However, the higher recall comes with a cost of more false positives (lower precision), on which TRATTO does not incur as often thanks to the fine-tuning performed, which allows it to evaluate more precisely whether an oracle should be generated or not, and what the shape or the oracle should be. This is also illustrated with the *F1-score*, which well summarizes the improvement of the token-by-token neuro-symbolic approach of TRATTO over the symbolic approach of Jdoctor and the neural approach of GPT4. The precision and recall of the approaches vary greatly across projects, thus confirming the dependency of the approaches on the quality of comments and code.

Table 2 provides futher details for each project and approach. The table reports the number of methods (column M), the number of oracles in the ground truth (O), the number of non-oracle (negative) instances (NO), and the true and false predictions (TP, TN, FP, FN) for all approaches

Table 1. Accuracy (A), precision (P), recall (R), and F1-score ($F1$) for all approaches on ground-truth dataset.

		closure-compiler	commons-cli	commons-codec	commons-compress	commons-csv	commons-jxpath	commons-lang	gson	jackson-core	jackson-databind	jackson-dataformat	jfree-chart	joda-time	jsoup	mockito	Total
Tratto	A	56%	83%	61%	74%	91%	62%	75%	80%	68%	58%	67%	77%	81%	60%	67%	**73%**
	P	N/A	60%	29%	82%	88%	N/A	61%	70%	43%	33%	N/A	90%	83%	50%	N/A	**72%**
	R	0%	75%	24%	53%	97%	0%	67%	70%	33%	3%	0%	60%	78%	5%	0%	52%
	F1	N/A	67%	26%	64%	92%	N/A	64%	70%	37%	6%	N/A	72%	80%	8%	N/A	**61%**
Jdoctor	A	56%	72%	56%	73%	79%	62%	60%	64%	67%	61%	67%	47%	59%	57%	67%	61%
	P	N/A	100%	33%	100%	96%	N/A	0%	6%	33%	100%	N/A	N/A	74%	0%	N/A	62%
	R	0%	16%	4%	41%	68%	0%	0%	7%	11%	3%	0%	0%	33%	0%	0%	**16%**
	F1	N/A	29%	7%	58%	79%	N/A	N/A	6%	17%	6%	N/A	N/A	46%	N/A	N/A	**25%**
GPT4	A	18%	26%	37%	42%	55%	5%	53%	19%	47%	35%	50%	35%	53%	41%	50%	**40%**
	P	12%	7%	20%	27%	40%	2%	31%	3%	22%	13%	0%	27%	42%	23%	33%	**24%**
	R	100%	100%	92%	100%	92%	100%	88%	60%	100%	100%	N/A	92%	96%	67%	100%	**89%**
	F1	22%	12%	32%	42%	56%	4%	46%	6%	36%	24%	N/A	42%	58%	34%	50%	37%

and projects, as well as the totals (last row). For each metric, we report both the total number of predictions as well as the percentage over the total number of instances ($O + NO = 389 + 496 = 885$). Note that the sum of $TP + TN + FP + FN$ may go over 885 as any approach can generate any arbitrary number of false positives.

Row *Total* of Table 2 highlights in green the best and in red the worst performance of the three approaches. Tratto generates the highest rate of correct oracles (total true positive rate 21%), and avoids generating oracles that shall not be generated in a good number of cases (total true negative rate 52%, just below the best result of Jdoctor, 54%). Tratto performs well also in terms of false alarms and missed oracles, with low false positive/negative rates, i.e., 8% and 19%, respectively and overall. Jdoctor performs sightly better than Tratto in terms of true negative (54% vs. 52%) and false positive rates (4% vs. 8%), however, it performs worst among the three approaches in terms of both true positive (7%) and false negative rates (35%). GPT4 presents an excellent false negative rate (only 2%) however with poor true positive (18%), true negative (22%) and false positive rates (58%). The true and false positive and negative rates confirm the best performance of Tratto among the three approaches, as well summarized by the best *F1-score* in Table 1.

Overall, Tratto generates 186 out of 389 oracles in the ground truth and demonstrates a good capability to discern when an oracle should be generated, with a relatively small number of wrong oracles (72 out of 885 predictions). Thus, Tratto can significantly reduce the effort of manually generating oracles without a big overhead for identifying and discarding wrong oracles.

Tratto generates three times more correct oracles than Jdoctor (186 vs. 58). The result is not surprising: Jdoctor exploits classic natural language processing and semantic matching, and as such, it works well in the presence of precise comments in natural language, however it does not process well the imprecise comments that often occur in Javadoc documentation. The neuro-symbolic approach of Tratto is much more tolerant with respect to the precision of the comments in natural language, and handles many more cases than Jdoctor. The results highly depend on the quality of the comments and code. Jdoctor generates a fair number of oracles for `commons-csv` (23 oracles while Tratto generates 30), but does not generate any oracle for `jfree-chart`, while Tratto generates 26 correct oracles. GPT4 infers 217 correct oracles, a relative small increment with respect to Tratto (186 correct oracles). This reflects the huge difference in the size of the respective neural components (GPT4 features hundreds of billions of parameters against the 7 billion parameters of Code Llama, the reference model of Tratto). GPT4 generates 701 false positives, almost 10 times more than Tratto (72), which accounts for more than half of all the predictions (58%). The significant amount of false positives showcases the main difference between the pure neural approach of GPT4 and the neuro-symbolic approach of Tratto: the symbolic component of Tratto discriminates valid from invalid results, while its neural component is specifically trained for discerning the possibility to generate and oracle (*Oracle evaluator*), thus Tratto generates less valid results than GPT4 but also a much more limited number of invalid results, while GPT4 does

Table 2. True/false positives/negatives ($TP/TN/FP/FN$) for all approaches on ground-truth dataset.

Project	M	O	NO	Tratto				Jdoctor				GPT4			
				TP	TN	FP	FN	TP	TN	FP	FN	TP	TN	FP	FN
closure-compiler	3	4	5	0 (0%)	5 (56%)	0 (0%)	4 (44%)	0 (0%)	5 (56%)	0 (0%)	4 (44%)	2 (12%)	1 (6%)	14 (0%)	0 (82%)
commons-cli	6	6	12	3 (16%)	12 (67%)	2 (6%)	1 (11%)	1 (5%)	12 (67%)	0 (0%)	5 (28%)	1 (5%)	4 (21%)	14 (74%)	0 (0%)
commons-codec	19	24	33	4 (7%)	32 (54%)	10 (22%)	13 (17%)	1 (2%)	31 (54%)	2 (4%)	23 (40%)	11 (15%)	16 (22%)	45 (62%)	1 (1%)
commons-compress	12	17	20	9 (24%)	19 (50%)	2 (5%)	8 (21%)	7 (19%)	20 (54%)	0 (0%)	10 (27%)	12 (21%)	12 (21%)	33 (58%)	0 (0%)
commons-csv	16	35	23	30 (52%)	23 (40%)	4 (7%)	1 (2%)	23 (39%)	23 (39%)	1 (2%)	11 (18%)	24 (28%)	23 (27%)	36 (42%)	2 (2%)
commons-jxpath	4	5	8	0 (0%)	8 (62%)	0 (0%)	5 (38%)	0 (0%)	8 (62%)	0 (0%)	5 (38%)	1 (2%)	2 (4%)	49 (94%)	0 (0%)
commons-lang	53	64	103	37 (22%)	89 (53%)	24 (14%)	18 (11%)	0 (0%)	101 (61%)	2 (1%)	64 (38%)	38 (20%)	62 (33%)	83 (44%)	5 (3%)
gson	27	30	51	19 (23%)	46 (57%)	8 (10%)	8 (10%)	1 (1%)	51 (63%)	16 (20%)	13 (16%)	3 (3%)	18 (16%)	88 (79%)	2 (2%)
jackson-core	10	10	20	3 (10%)	18 (58%)	4 (13%)	6 (19%)	1 (3%)	19 (63%)	2 (7%)	8 (27%)	5 (15%)	11 (32%)	18 (53%)	0 (0%)
jackson-databind	24	30	48	1 (1%)	42 (57%)	2 (3%)	29 (39%)	1 (1%)	46 (60%)	0 (0%)	29 (39%)	13 (10%)	33 (25%)	84 (65%)	0 (0%)
jackson-dataformat	1	1	2	0 (0%)	2 (67%)	0 (0%)	1 (33%)	0 (0%)	2 (67%)	0 (0%)	1 (33%)	0 (0%)	2 (50%)	2 (50%)	0 (0%)
jfree-chart	25	46	40	26 (0%)	40 (47%)	3 (3%)	17 (20%)	0 (0%)	40 (47%)	0 (0%)	46 (53%)	34 (24%)	17 (12%)	90 (62%)	3 (2%)
joda-time	40	71	63	52 (39%)	56 (42%)	11 (8%)	15 (11%)	23 (17%)	56 (42%)	8 (6%)	47 (35%)	50 (33%)	31 (20%)	70 (46%)	2 (1%)
jsoup	33	45	66	2 (2%)	65 (59%)	2 (2%)	42 (38%)	0 (0%)	64 (57%)	4 (3%)	45 (40%)	22 (16%)	36 (25%)	73 (51%)	11 (8%)
mockito	1	1	2	0 (0%)	2 (67%)	0 (0%)	1 (33%)	0 (0%)	2 (67%)	0 (0%)	1 (33%)	1 (25%)	1 (25%)	2 (50%)	0 (0%)
Total	274	389	496	186 (**21%**)	459 (52%)	72 (8%)	169 (19%)	58 (**7%**)	480 (**54%**)	35 (**4%**)	312 (**35%**)	217 (18%)	269 (**22%**)	701 (**58%**)	26 (**2%**)

not check the validity and compilability of the results, and often generates oracles in cases where none should be generated. The large amount of false positives of GPT4 greatly reduces its practical applicability since it requires massive human effort to prune the results.

GPT4 produces very few false negatives (26, 2% of the total predictions), while Tratto and Jdoctor do not generate any oracles for 169 and 312 oracles in the ground truth (19% and 35% of the total predictions), respectively. The good performance of GPT4 in terms of false negatives balances the bad results in terms of false positives: GPT4 mostly always attempts to generate oracles. As a consequence, it generates lots of wrong oracles (false positives) and misses very few cases (false negatives). On the contrary, both Tratto and Jdoctor identify those cases where no oracle should be generated, and generate less wrong oracles. The overall performance of the approaches is well reflected by the combination of false predictions (positives and negatives): GPT4 generates a total of 727 false results (701 FP + 26 FN), Jdoctor 347 (35 + 312) and Tratto only 241 (72 + 169).

The second `@throws` tag in Listing 6 (line 6) well exemplifies the capability of Tratto to infer correct oracles from Javadoc tags in terms of exceptional postconditions that both Jdoctor and GPT4 fail to generate. The `@throws` tag contains an implicit reference (line 6, *"this method is called on a closed result set"*) to a method (`isClosed`) of a class (`ResultSet`) of an external library (`java.sql`): Tratto's symbolic module retrieves the contextual information related to the external class from the signature of the method (line 9, `resultSet` method parameter) and feeds the neural model with additional information to produce the correct axiomatic oracle "`resultSet.isClosed();`". Jdoctor cannot infer an oracle from the Javadoc tag, since it falls outside the set of rules and patterns, resulting in a *false negative*. GPT4 generates the precondition "`resultSet != null;`", claiming that the parameter `resultSet` must not be null, even if neither the documentation nor the contextual information indicate that the method cannot accept null values in input. GPT4 generates also three exceptional postconditions ("`resultSet == null ? NullPointerException;`", "`IOException may be thrown;`" and "`SQLException may be thrown if resultSet is closed or if there is a database access error`") that are either wrong or not compilable. The example highlights how a pure neural model struggles to infer non-trivial oracles, may generate non-compilable oracles since it lacks a precise grammar, and generates a non-negligible number of *false positives*.

A manual inspection of the oracles generated with Tratto and Jdoctor shows that there is only one oracle that Jdoctor generates and Tratto does not. This was an exceptional oracle stating the need of a method parameter to be of a certain type (via the `instanceof` operator). We hypothesize that Tratto's training set did not include enough oracles of this kind and thus the *Oracle evaluator* judged no need to generate an oracle in this case. On the other hand, Tratto generates significantly more oracles than Jdoctor, as the symbolic matching of Jdoctor often overlooks similar cases. For example, Tratto successfully generates exceptional oracles "`source == null;`" and "`bigInteger == null;`" from the Javadoc comments *"the parameter passed to this method is null"* in Listing 7 (lines 14-15) and *"@throws NullPointerException if null is passed in"* in Listing 8 (line 8), respectively. Jdoctor generates only the first oracle, and cannot generate the second, despite the similarity of the two Javadoc comments. The neural component of Tratto enhances Tratto's ability to identify interesting oracles and explains the significantly higher recall of Tratto over Jdoctor.

```java
/**
* Prints headers for a result set based on its metadata.
*
* @param resultSet The ResultSet to query for metadata.
* @throws IOException If an I/O error occurs.
* @throws SQLException If a database access error occurs or this method is called on a closed result set.
* @since 1.9.0
*/
public synchronized void printHeaders(final ResultSet resultSet) throws IOException, SQLException {
  printRecord((Object[]) format.builder().setHeader(resultSet).build().getHeader());
}
```

Listing 6. Documentation and implementation of method printHeaders from CSVPrinter.

```java
 1  /**
 2  * Decodes an "encoded" Object and returns a
 3  * "decoded" Object. Note that the implementation of
 4  * this interface will try to cast the Object
 5  * parameter to the specific type expected by a
 6  * particular Decoder implementation. If {@link
 7  * ClassCastException} occurs this decode method will
 8  * throw a DecoderException.
 9  *
10  * @param source the object to decode
11  * @return a 'decoded" object
12  * @throws DecoderException a decoder exception can
13  * be thrown for any number of reasons. Some good
14  * candidates are that the parameter passed to this
15  * method is null, a param cannot be cast to the
16  * appropriate type for a specific encoder.
17  */
18  Object decode(Object source) throws DecoderException;
```

Listing 7. Documentation and implementation of method decode from Decoder.

```java
 1  /**
 2  * Encodes to a byte64-encoded integer according to
 3  * crypto standards such as W3C's XML-Signature.
 4  *
 5  * @param bigInteger a BigInteger
 6  * @return A byte array containing base64 character
 7  * data
 8  * @throws NullPointerException if null is passed in
 9  * @since 1.4
10  */
11  public static byte[] encodeInteger(final BigInteger
12      bigInteger) {
13    Objects.requireNonNull(bigInteger, "bigInteger");
14    return encodeBase64(toIntegerBytes(bigInteger),
15          false);
16  }
```

Listing 8. Documentation and implementation of method encodeInteger from Base64.

Answer to RQ$_3$: Tratto achieves an F1-score of 61% in generating axiomatic oracles, significantly higher than Jdoctor (25%) and GPT4 (37%). Tratto can generate a high number of oracles (186, 3× more than Jdoctor) incurring in few false positives (72, 10× less than GPT4).

4.4 RQ$_4$: Robustness to Documentation Variations

RQ$_4$ evaluates the robustness of Tratto, Jdoctor and GPT4 in terms of their ability to generate axiomatic oracles with varying degrees of documentation quality. We select all oracles that all three approaches successfully generate (RQ$_3$) and systematically generate variations of their associated documentation. We then compute the proportion of oracles that the approaches can still generate with the modified documentation.

There are 55 oracles that all three approaches successfully generate from eight projects from the ground-truth dataset. For each oracle, we identify the Javadoc tag based on which the oracle was generated, and we generate four variations of it. We automate the process and make it systematic, by asking ChatGPT to generate alternative versions of the Javadoc tag with the following criteria: (i) replacing words with synonyms or rephrasing sentences while keeping the same meaning; (ii) changing the order of the sentence; (iii) introducing grammatical mistakes or typos; and (iv) making the sentence less explicit. Listing 9 shows some examples of the generated variations.

We manually check the correctness of the oracles that the approaches generate for each of the 220 variations (55 oracles × 4 variations). Table 3 shows the number and percentage of oracles that each approach correctly generates. Tratto generates about 90% of correct oracles for the documentation variations, about the same as GPT4, despite relying on a much smaller model. Indeed, the performance of Tratto is similar to GPT4 for all types of variations except for less explicit descriptions. This is expected since these do not explicitly provide information regarding oracles (e.g., line 6 in Listing 9). Jdoctor is the least robust approach, with only 22% correct oracles for the variations, and no oracles at all for less explicit descriptions.

Table 3. Robustness to documentation variations.

Approach	Synonyms	Order	Typos	Explicitness	Total
Tratto	53 (96%)	54 (98%)	54 (98%)	34 (62%)	195 (89%)
Jdoctor	16 (29%)	29 (53%)	4 (7%)	0 (0%)	49 (22%)
GPT4	52 (95%)	54 (98%)	53 (96%)	49 (89%)	208 (95%)

```
1 (object == null) == false; // Generated oracle
2 @param object the object to convert, must not be null // Original Javadoc tag
3 @param object the item to transform, should not be null // Synonyms or rephrasing
4 @param object Must not be null, this is the object that needs conversion. // Changed order
5 @param object the obect to convirt, musn't be nul // Grammatical mistakes or typos
6 @param object an element to be used, should be valid // Less explicit
```

Listing 9. Rephrased Javadoc tags generated with ChatGPT.

4.5 RQ$_5$: Application to Software Testing

We address RQ$_5$ with an exploratory study on the impact of the automatically generated oracles on the mutation score [29] of test suites. We developed a script to automatically insert the generated oracles into test cases that contain a call to a method for which an oracle was generated. Oracles that make a test case fail are simply discarded, although a further check on them may discover bug-revealing tests, further strengthening the usefulness of the oracles. Then, the mutation score is computed with the PIT mutation testing tool [15]. We measure the impact of the oracles as the difference between the mutation score of the test suite with and without the oracles. We excluded GPT4 from this experiment due to the large amount of non-compilable oracles it generates (552 out of 1,213 in RQ$_3$), which makes it impossible to automate the insertion of oracles into test suites.

We generated test suites with EvoSuite [25] for the 10 classes of each project from the ground-truth dataset for which we generated oracles as part of RQ$_3$. We were unable to run EvoSuite on five projects—`closure-compiler`, `jackson-core`, `jackson-databind`, `jackson-dataformat` and `mockito`—due to incompatibility issues. We did not consider three projects for which Jdoctor did not generate oracles from the ground-truth dataset—`commons-lang`, `jfree-chart` and `jsoup`—and a project for which neither Tratto nor Jdoctor generated oracles—`commons-jxpath`—since, without oracles, the test suite and consequently the mutation score do not change. By focusing on the projects for which *both* Tratto and Jdoctor generate oracles, we aim to ensure a fair comparison.

We evaluated the improvement achieved with the automatically generated oracles upon test suites with implicit oracles only, and with both implicit and regression oracles. The latter represent the default test suites generated with EvoSuite, while the former entail a more realistic scenario, where regression oracles are not available, or it cannot be assumed that they are correct. To create these test suites, we simply wrap the assertions generated by EvoSuite into try-catch blocks.

Table 4 shows the mutation score of the test suites with Evosuite, Tratto and Jdoctor, with and without regression oracles. The oracles generated by Tratto improve the mutation score with respect to Evosuite's implicit oracles for 5 out of 6 projects, from 2% (`commons-compress`) up to 17% (`commons-codec`), with an average increase of 7% across all projects. The oracles improve the mutation score also in the presence of Evosuite's regression oracles on 3 out of 6 projects with an average increase of 1%. This lower increase was expected since EvoSuite's regression oracles make concrete assumptions of expected behavior and thus may be more effective than axiomatic oracles in this context, although not so widely applicable nor necessarily correct. Table 4 also highlights the worse performance of Jdoctor compared to TRATTO, as it improved the mutation score of 4 out of 6 test suites with implicit oracles (1% average increase vs. 7% of TRATTO) and in no cases for test suites with regression oracles.

Despite the modest results in this exploratory study, two things are worth noting. First, TRATTO can only enhance test suites, not worsen them, as the new oracles generated may reveal actual bugs or increase their mutation score in the best-case scenario. Second, software testing is just one possible application of the oracles generated by TRATTO, as they may be useful in other contexts such as program comprehension, requirements specification or runtime verification.

Table 4. Mutation score of test suites.

Project	Implicit oracles			Implicit/regression oracles		
	EvoSuite	TRATTO	Jdoctor	EvoSuite	TRATTO	Jdoctor
commons-cli	21%	**28%**	22%	61%	61%	61%
commons-codec	42%	**59%**	46%	71%	**75%**	71%
commons-compress	20%	**22%**	20%	41%	41%	41%
commons-csv	5%	5%	5%	13%	13%	13%
gson	22%	**30%**	24%	67%	**68%**	67%
joda-time	38%	**47%**	39%	78%	**79%**	78%
Total	25%	**32%**	26%	55%	**56%**	55%

5 Threats to Validity

Internal validity. The datasets used in our experiments may contain wrong instances, since they are based on an automatically generated dataset [6]. We mitigate this threat by manually inspecting and fixing or discarding wrong instances. Manually generated or modified instances were inspected by two authors to minimize bias, and conflicts were solved via open discussion. Indeed, all processes involving manual analysis, including the generation of the ground-truth dataset and the analysis of the predictions by TRATTO, Jdoctor and GPT4, were performed by two authors. All our datasets, results, and tool implementations are available as open source in our replication package [2].

In answering RQ_1 and RQ_2, we measure the differences across the techniques (code models and ablated approaches) in terms of the accuracy in generating the next oracle token. This does not take into account equivalent valid oracles that the model can generate. For example, the oracle "`param < 0;`" is equivalent to "`(param < 0);`", thus both tokens '`(`' and '`param`' are valid when the partial oracle is empty. However, only '`param`' is deemed as correct. The significant difference between the accuracies of the three models in RQ_1 relieves the risk of missing the best LLM for TRATTO. As for RQ_2, we noticed that this phenomenon occurred in both cases (for TRATTO and the ablated approaches), partially alleviating this threat to the validity of the results. In RQ_3 and RQ_4 we manually analyzed all oracles generated to fully neutralize this threat and properly compare TRATTO against the state-of-the-art approaches, while RQ_5 is not affected by this threat.

In generating the ground-truth dataset, we decided to keep only those methods featuring at least one oracle, otherwise the dataset would be extremely unbalanced, containing 389 positive samples (oracles) and 6,485 negative samples (units for which no oracles can be extracted). Furthermore, this would make the manual analysis extremely costly, due to the myriad of false positives generated by GPT4. We did analyze the results for TRATTO and Jdoctor considering also the 6,485 negative samples and found that the recall for both remained in similar levels (49% for TRATTO and 19% for Jdoctor) while precision decreased (49% for TRATTO and 57% for Jdoctor). This was expected due to the higher amount of negative samples, making both approaches incur in more false positives.

External validity. Our experiments with Java methods do not prove the generalizability of TRATTO to other programming languages. Nevertheless, the approach can be easily adapted to other languages since it merely relies on source code and documentation of inputs and outputs. The symbolic component simply needs a language grammar and context restrictions, while for the neural module an ML model pre-trained on the target programming language is sufficient.

Tratto takes about five times more to analyze a Java class and generate oracles for it, compared to Jdoctor. This is partly because Tratto generates more oracles. Generating oracles token by token causes an overhead, but acceptable for the improvement achieved. Scalability can be addressed as Tratto's neural module is decoupled from the symbolic module, and the communication is handled through a REST API [21, 46]. In practice, this means that the neural module could be deployed in a more powerful server, leveraging a bigger code model, to speed up oracle generation.

6 Related Work

Our work combines symbolic and neural techniques to generate axiomatic oracles. Neuro-symbolic approaches have been studied for some SE problems, like code completion [5, 32, 49] or program synthesis [9, 43]. To the best of our knowledge, Tratto is the first neuro-symbolic approach tailored to the oracle problem. Next, we discuss both symbolic and neural techniques for oracle generation.

6.1 Symbolic Approaches

Symbolic approaches leverage static and dynamic program analysis and natural language processing (NLP) to understand the semantics of software systems and produce semantically relevant oracles.

The seminal approaches Daikon [20] and Dysy [17] on invariant mining execute a program on a collection of inputs against a collection of potential invariants. The accuracy of the invariants inferred with these methods depends on both the quality and completeness of the test cases and the collection of potential invariants provided. Evospex [37] overcomes the limitations of the previous techniques by exercising the unit under test through its APIs to generate valid pre and post states, without requiring any specification or test. The valid pre and post states undergo mutations that lead to corresponding invalid pre and post states and a genetic algorithm infers valid postconditions guided by the valid/invalid states. GAssert [52] improves automatically inferred assertion oracles, by reducing false positives and negatives with an evolutionary algorithm. All these approaches derive test oracles, by both relying on the execution of the current version of a program and generating regression oracles that cannot detect if the bug is already present in the program.

Text-driven specification mining methods exploit NLP, pattern, semantic, and syntax matching to generate test oracles from code comments and text documentation. ALICS [42] mines code contracts in the form of pre- and postconditions from code comments. ALICS uses NLP-based pattern matching with an application domain specific dictionary which hardly generalizes. @tComment [50] infers null-deference properties of parameters from Javadoc comments, with natural language patterns and heuristics. The patterns are quite narrow and do not generalize to other exception types.

Toradocu [27], JDoctor [6], MeMo [8] and CaMeMa [7] use natural language parsing and lexical matching to generate preconditions, normal and exceptional postconditions, metamorphic relations, and temporal properties, from Javadoc comments. These approaches can infer oracles when code comments fit the defined patterns, but do not generalize when comments fall outside those. These approaches are not applicable when code comments are unavailable. We discussed the experimental comparison of Tratto with Jdoctor, the most complete of this family of approaches, in $RQ_{3,4,5}$.

6.2 Neural Approaches

Recently, there has been an upsurge of works that exploit DL for SE and software testing, with many applications of neural-based transformers and transfer learning to automatically generate semantically relevant assertion oracles and entire unit tests, overcoming the limitations of symbolic techniques even in the absence of precise patterns or entire docstrings.

ATLAS [59] leverages a recurrent neural network (RNN) to generate assertion oracles from a unit under test and a test prefix. ATLAS relies only on the source code and ignores the documentation. It solely targets normal postconditions and cannot generate oracles for preconditions or exceptional behavior. Subsequently, Tufano et al. [54] proposed AthenaTest, which outperformed ATLAS by replacing the RNN with transformer models pre-trained on natural language and code. This was the first noteworthy approach to generate test cases including both test prefixes and oracles, considering both the unit implementation and its context, such as the surrounding class and method signature. AthenaTest does not consider the information provided in the code documentation.

Dinella et al. [19] redesigned the oracle generation problem as a two-step neural ranking procedure. Their approach, TOGA, exploits two pre-trained models fine-tuned on the task of discerning among normal and exceptional behaviors to generate test oracles. The exceptional oracle classifier infers whether a test prefix should throw an exception, from the unit under test, the docstring (if available) and a test prefix. The assertion oracle ranker generates an assertion oracle conforming to a grammar of candidate assertions, which enforces syntactic and type correctness. The wide applicability and high flexibility of neural approaches comes with a cost in terms of accuracy. TOGA mitigates the issue by restricting the approach to a subset of plausible and syntactically valid oracles. Hossain et al. [28] analyzed the current neural-based test oracle generation approaches (including TOGA) and highlighted how the inferred assertions still exhibit high false positive rates that threaten their practical usefulness. Our proposed approach tackles this limitation.

All neural approaches discussed generate concrete assertions as normal and exceptional postconditions. Tratto is the first attempt to successfully generate axiomatic oracles, including preconditions, to invalidate tests when input values are not satisfied, thus mitigating false positives.

7 Conclusion

In this paper, we propose Tratto, a neuro-symbolic approach to automatically generate axiomatic oracles from commonly available information: documentation and code. Axiomatic oracles in the form of pre- and postconditions are particularly useful as a complement to automatically generated test suites, as preconditions can rule out invalid inputs, reducing false positives, and postconditions can detect bugs, reducing false negatives. Moreover, axiomatic oracles are applicable to any test case, as they predicate on input and output variables and states.

Tratto entails a novel reformulation of the oracle generation problem, as a token-by-token generation approach. The symbolic module of Tratto effectively restricts the search space of the tokens that may be used to generate an oracle, while the neural component steers the generation process towards semantically relevant oracles. Our ablation studies confirm the contributions of both the symbolic and neural components over solely symbolic- or neural-based approaches. Tratto generates over three times more correct oracles than state-of-the-art symbolic approaches (Jdoctor [6]) while incurring in 10 times less false positives than neural approaches (GPT4 complemented with few-shot learning and Chain-of-Thought prompting). Also, Tratto is significantly robust to the quality of documentation, and it can help improve the quality of existing test suites.

Our future research agenda will focus on applying neuro-symbolic techniques to generating concrete assertions and complete test cases.

Data Availability

Our replication package includes the source code of the scripts and programs developed, the data generated in the experiments (including datasets) and instructions on how to reuse the material for further research. The artifact can be downloaded from [2].

Acknowledgments

This research was supported in part by NSF grant CNS-2120070, DARPA contract FA8750-20-C-0226, and SNF projects A-Test Autonomic Software Testing (200021_215487) and PARSED (219294).

References

[1] [n.d.]. Dataset of procedure specifications by Blasi et al. https://github.com/albertogoffi/toradocu/tree/master/src/test/resources/goal-output.

[2] Anonymous. 2025. [Replication Package] Tratto: A Neuro-Symbolic Approach to Deriving Axiomatic Test Oracles. https://anonymous.4open.science/r/tratto-replication-package-31F6.

[3] Sergio Antoy and Dick Hamlet. 2000. Automatically Checking an Implementation against Its Formal Specification. *IEEE Transactions on Software Engineering* 26, 1 (2000), 55–69.

[4] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.

[5] Vitaliy Bibaev, Alexey Kalina, Vadim Lomshakov, Yaroslav Golubev, Alexander Bezzubov, Nikita Povarov, and Timofey Bryksin. 2022. All you need is logs: improving code completion by learning from anonymous IDE usage logs. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 1269–1279. https://doi.org/10.1145/3540250.3558968

[6] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating Code Comments to Procedure Specifications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '18)*. ACM.

[7] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2022. Call Me Maybe: Using NLP to Automatically Generate Unit Test Cases Respecting Temporal Constraints. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 19:1–19:11. https://doi.org/10.1145/3551349.3556961

[8] Arianna Blasi, Alessandra Gorla, Michael D Ernst, Mauro Pezze, and Antonio Carzaniga. 2021. MeMo: Automatically identifying metamorphic relations in Javadoc comments for test automation. *Journal of Systems and Software* 181 (2021), 111041.

[9] José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. 2023. FlashFill++: Scaling programming by example by cutting to the chase. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 952–981.

[10] T. Y. Chen, S. C. Cheung, and S. M. Yiu. 1998. *Metamorphic testing: A new approach for generating next test cases*. Technical Report HKUST-CS98-01. HKUST Department of Computer Science, Hong Kong.

[11] Betty HC Cheng and Joanne M Atlee. 2007. Research directions in requirements engineering. *Future of software engineering (FOSE'07)* (2007), 285–303.

[12] Yoonsik Cheon and Gary T. Leavens. 2002. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '02)*. 231–255.

[13] Yoonsik Cheon and Gary T. Leavens. 2002. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP 2002 — Object-Oriented Programming, 16th European Conference*. Málaga, Spain, 231–255.

[14] Koen Claessen and John Hughes. 2000. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP 2000: Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*. Montreal, Canada, 268–279.

[15] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: a practical mutation testing tool for Java. In *Proceedings of the 25th international symposium on software testing and analysis*. 449–452.

[16] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. 2009. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 35, 5 (2009), 684–702.

[17] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: Dynamic Symbolic Execution for Invariant Inference. In *Proceedings of the International Conference on Software Engineering (ICSE '08)*. ACM, 281–290.

[18] J. D. Day and J. D. Gannon. 1985. A Test Oracle Based on Formal Specifications. In *Proceedings of the Conference on Software Development Tools, Techniques, and Alternatives (SOFTAIR '85)*. 126–130.

[19] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu Lahiri. 2022. TOGA: A Neural Method for Test Oracle Generation. In *ICSE 2022*. ACM. https://www.microsoft.com/en-us/research/publication/toga-a-neural-method-for-test-oracle-generation/

[20] 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), 35–45.

[21] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph. D. Dissertation.

[22] George Fink and Matt Bishop. 1997. Property-based testing: A new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes* 22, 4 (July 1997), 74–80.

[23] Apache Software Foundation. 2023. Apache Commons Collections: The Apache Commons Collections Library. https://commons.apache.org/proper/commons-collections/. Accessed: 2024-06-07.

[24] Apache Software Foundation. 2023. Apache Commons Math: The Apache Commons Mathematics Library. https://commons.apache.org/proper/commons-math/. Accessed: 2024-06-07.

[25] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, 416–419.

[26] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

[27] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic Generation of Oracles for Exceptional Behaviors. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '16)*. ACM, 213–224.

[28] Soneya Binta Hossain, Antonio Filieri, Matthew B. Dwyer, Sebastian G. Elbaum, and Willem Visser. 2023. Neural-Based Test Oracle Generation: A Large-Scale Evaluation and Lessons Learned. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 120–132. https://doi.org/10.1145/3611643.3616265

[29] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (September 2011), 649–678.

[30] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '14)*. ACM, 437–440.

[31] Teemu Kanstren. 2009. Program comprehension for user-assisted test oracle generation. In *2009 Fourth International Conference on Software Engineering Advances*. IEEE, 118–127.

[32] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code Prediction by Feeding Trees to Transformers. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 150–162. https://doi.org/10.1109/ICSE43902.2021.00026

[33] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *The journal of logic and algebraic programming* 78, 5 (2009), 293–303.

[34] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. StarCoder 2 and The Stack v2: The Next Generation. arXiv:2402.19173 [cs.SE] https://arxiv.org/abs/2402.19173

[35] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2023. Using Transfer Learning for Code-Related Tasks. *IEEE Trans. Softw. Eng.* 49, 4 (apr 2023), 1580–1598. https://doi.org/10.1109/TSE.2022.3183297

[36] Quinn McNemar. 1947. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika* 12, 2 (1947), 153–157.

[37] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo Frias. 2021. EvoSpex: An evolutionary algorithm for learning postconditions. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1223–1235.

[38] OpenAI. 2023. ChatGPT: An AI Language Model. https://www.openai.com/chatgpt. Accessed: 2024-06-07.

[39] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). https://doi.org/10.48550/ARXIV.2303.08774 arXiv:2303.08774

[40] OpenAI. 2024. GPT-4 | OpenAI. https://openai.com/index/gpt-4/. Accessed: 2024-10-07.

[41] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the International Conference on Software Engineering (ICSE '07)*. ACM, 75–84.

[42] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring method specifications from natural language API descriptions. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 815–825.

[43] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-symbolic program synthesis. In *International Conference on Learning Representations*.

[44] Dennis K. Peters and David Lorge Parnas. 1998. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering* 24, 3 (3 1998), 161–173.

[45] Mauro Pezzè and Cheng Zhang. 2015. Automated Test Oracles: A Survey. In *Advances in Computers*. Vol. 95. Elsevier, 1–48.

[46] Leonard Richardson, Mike Amundsen, and Sam Ruby. 2013. *RESTful Web APIs*. O'Reilly Media, Inc.

[47] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL] https://arxiv.org/abs/2308.12950

[48] David Saff. 2007. Theory-infected: Or how I learned to stop worrying and love universal quantification. In *OOPSLA Companion: Object-Oriented Programming Systems, Languages, and Applications*. Montreal, Canada, 846–847.

[49] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. 2021. Fast and Memory-Efficient Neural Code Completion. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 329–340. https://doi.org/10.1109/MSR52588.2021.00045

[50] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST '12)*. IEEE Computer Society, 260–269.

[51] CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, Luke Vilnis, Mateo Wirth, Paul Michel, Peter Choy, Pratik Joshi, Ravin Kumar, Sarmad Hashmi, Shubham Agrawal, Zhitao Gong, Jane Fine, Tris Warkentin, Ale Jakse Hartman, Bin Ni, Kathy Korevec, Kelly Schaefer, and Scott Huffman. 2024. CodeGemma: Open Code Models Based on Gemma. arXiv:2406.11409 [cs.CL] https://arxiv.org/abs/2406.11409

[52] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2020. Evolutionary Improvement of Assertion Oracles. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE '20)*. ACM, 1178–1189.

[53] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized unit tests. In *ESEC/FSE 2005: Proceedings of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Lisbon, Portugal, 253–262.

[54] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2021. Unit Test Case Generation with Transformers and Focal Context. arXiv:2009.05617 [cs.SE]

[55] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2022. Generating Accurate Assert Statements for Unit Test Cases Using Pretrained Transformers. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test* (Pittsburgh, Pennsylvania) *(AST '22)*. Association for Computing Machinery, New York, NY, USA, 54–64. https://doi.org/10.1145/3524481.3527220

[56] Chunhui Wang, Fabrizio Pastore, and Lionel Briand. 2018. Oracles for testing software timeliness with uncertainty. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2018), 1–30.

[57] Xin Wang, Ji Wang, and Zhi-Chang Qi. 2004. Automatic Generation of Run-Time Test Oracles for Distributed Real-Time Systems. In *Formal Techniques for Networked and Distributed Systems – FORTE 2004*, David de Frutos-Escrig and Manuel Núñez (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 199–212.

[58] Yaqing Wang, Quanming Yao, James T. Kwok, and Lionel M. Ni. 2021. Generalizing from a Few Examples: A Survey on Few-shot Learning. *ACM Comput. Surv.* 53, 3 (2021), 63:1–63:34. https://doi.org/10.1145/3386252

[59] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1398–1409.

[60] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 24824–24837. https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf