A Hybrid Approach for Inference between Behavioral Exception API Documentation and Implementations, and Its Applications

Hoan Anh Nguyen
Amazon
Washington, USA
nguyenanhhoan@gmail.com

Hung Dang Phan
Iowa State University
Iowa, USA
hungphd@iastate.edu

Samantha Khairunnesa
Bradley University
Illinois, USA
skhairunnesa@bradley.edu

Son Van Nguyen
University of Texas at Dallas
Texas, USA
sonnguyen@utdallas.edu

Aashish Yadavally
University of Texas at Dallas
Texas, USA
aashish.yadavally@utdallas.edu

Shaohua Wang
New Jersey Institute of Technology
New Jersey, USA
davidshwang@ieee.org

Hridesh Rajan
Iowa State University
Iowa, USA
hridesh@iastate.edu

Tien N. Nguyen
University of Texas at Dallas
Texas, USA
tien.n.nguyen@utdallas.edu

ABSTRACT

Automatically producing behavioral exception (BE) API documentation helps developers correctly use the libraries. The state-of-the-art approaches are either rule-based, which is too restrictive in its applicability, or deep learning (DL)-based, which requires large training dataset. To address that, we propose StatGen, a novel hybrid approach between statistical machine translation (SMT) and tree-structured translation to generate the BE documentation for any code and vice versa. We consider the documentation and source code of an API method as the two abstraction levels of the same intent. StatGen is specifically designed for this two-way inference, and takes advantage of their structures for higher accuracy.

We conducted several experiments to evaluate StatGen. We show that it achieves high precision (75% and 75%), and recall (81% and 84%), in inferring BE documentation from source code and vice versa. StatGen achieves higher precision, recall, and BLEU score than the state-of-the-art, DL-based baseline models. We show StatGen's usefulness in two applications. First, we use it to generate the BE documentation for Apache APIs that lack of documentation by learning from the documentation of the equivalent APIs in JDK. 44% of the generated documentation were rated as useful and 42% as somewhat useful. In the second application, we use StatGen to detect the inconsistency between the BE documentation and corresponding implementations of several JDK8 packages.

This work was done during the first author's postdoc program at Iowa State Univeristy.

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3560434

CCS CONCEPTS

• **Software and its engineering → Software verification**.

KEYWORDS

Behavioral Exception Specification; Hybrid Machine Translation; Inference between Documentation and Code

ACM Reference Format:
Hoan Anh Nguyen, Hung Dang Phan, Samantha Khairunnesa, Son Van Nguyen, Aashish Yadavally, Shaohua Wang, Hridesh Rajan, and Tien N. Nguyen. 2022. A Hybrid Approach for Inference between Behavioral Exception API Documentation and Implementations, and Its Applications. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3560434

1 INTRODUCTION

API documentation specifies the conditions that users must follow to integrate APIs into their codebase. Specifically, behavioral exception (BE) documentation indicates the circumstances under which an API throws an exception. Besides allowing users to understand the APIs better [54], API documentation on BE also enables automated tools to verify program properties [57]. However, the lack of specifications has hindered software library utilization [12]. To address that, many approaches have been proposed for automated API documentation generation. They can be classified into four categories: *program analysis* (PA), *mining software repositories* (MSR), *rule-based natural language processing (NLP)*, and *machine learning* (ML).

In *the PA direction*, dynamic analysis approaches [3–7, 15, 17, 28, 29, 33–37, 40, 46, 51, 52, 64] dynamically infer program properties by detecting data and temporal invariants. However, they require large test suites, which might be incomplete, leading to inaccurate results. In contrast, static analysis approaches [9, 14, 27, 30, 53, 63, 68] overestimate program behaviors and have high false positives.

Techniques in MSR leverage data mining to derive or check API usage specifications from existing repositories. They mine the call sites of the APIs and focus on the API usage or temporal orders, e.g.,

sets [31, 32], pairs [16, 61, 65] or sequences [58, 70] of co-occurring APIs, predicates [42, 50], or graphs of APIs [44, 48, 60, 67]. They do not focus on behavior exceptions/properties of the mined code.

Several rule-based, Natural language Processing (NLP) techniques [8, 18, 57, 71] analyze existing textual documentation to derive the pre-conditions, based on which they detect inconsistencies and directive defects in the corresponding source code. They use rules, patterns, lexical and semantic matching to derive pre/post-conditions to support testing. While these approaches do not require large training data, the extracted lexical and semantic rules are restricted by the pre-defined templates and not generalizable for BE documentation.

Recent advances in *Machine Learning* (ML) are also applied in specification mining. Phan *et al.* [47] leverage *phrase-based Statistical machine translation* (SMT) to derive BE specification from code. However, it treats code as texts without considering structure.

Other *advanced ML approaches* including *deep learning* (DL) models, e.g., tree-based transformer [56], dual-task learning [62], neural network machine translation [11, 22, 23], are used for the general purpose of code-to-text or text-to-code translations. However, these approaches cannot directly be extended for the translation between BE documentation and source code for the following reasons: first, they require a large amount of training data, which is not viable as software libraries often lack such specifications [42]; second, they are not specifically designed to exploit the structural information in (either or) both BE documentation sentences and source code.

In this work, we propose StatGen, a hybrid (structural and statistical) machine translation approach that supports bi-directional inference between BE documentation and its corresponding implementation. We consider an API method to possess two levels of abstraction: its implementation and documentation. Both serve a similar purpose, i.e., the BE documentation (in the natural language domain) outlines the behavior before or after the API method is called, and the respective BE part of source code (in the programming language domain) describes the exception behavior.

To overcome the aforementioned issues, we customize/modify an SMT model and enhance it with the structural information from both documentation and source code. Besides enabling learning in a low-data setting, this reduces the burden of mapping/translating the structures as in the sequence-based DL models. Moreover, it provides the flexibility over the rule-based NLP approaches in learning implicit translation rules in both directions.

The key idea in StatGen is to carry out translation via divide-and-conquer, in a syntax-directed fashion, that is similar in spirit to syntax-directed program editing to improve editing correctness. For code → BE documentation, we take advantage of well-structured source code to generate the clausal structure of a sentence in BE documentation. Next, for each syntactic code structure, we use phrase-based SMT to produce the translated clause in the documentation. Finally, the translated clauses are merged according to the clausal structure to produce the final BE documentation. For BE documentation → code, we first parse the sentences and normalize them into the clause trees that represent the conditions for the API stated in the documentation. We then generate the syntactic units in source code corresponding to those clause trees, and use the phrase-based SMT to produce the code for each clause. The final BE code is formed by merging the results for all clauses, based on the generated syntactic structure in source code.

Figure 1 Example of `java.lang.String`.

```
// @throws StringIndexOutOfBoundsException - if beginIndex is negative
// or larger than the length of this String object.
public String substring(int beginIndex) {
  if (beginIndex < 0) throw new StringIndexOutOfBoundsException(beginIndex);
  if (beginIndex > length) throw new StringIndexOutOfBoundsException(beginIndex); ...
}
```

StatGen's two-way inference supports the following usages: (1) if new API code does not have BE documentation, one could use StatGen to generate and use it as an initial point; (2) if an API has BE documentation, one could check the consistency between the implementation and the BE documentation by asking StatGen to generate the code from the BE documentation, then checking the code against the generated BE code. An inconsistency signals an out-of-date or imprecise documentation or source code. This could reduce potential misuses by alerting a developer in API usages.

We empirically evaluate StatGen to show that it achieves high precision (75% and 75%), and recall (81% and 84%), as inferring BE documentation from source code and vice versa, respectively. Our results also show that StatGen outperforms the state-of-the-art SMT [47], dual-task learner [62], tree-based transformer [56], and hybrid neural machine translation [22]. We also showed its usefulness in two applications in both directions Code-to-Doc and Doc-to-Code (we use the word *Doc* to refer to BE documentation and the word *Code* for the exception-throwing BE code). First, we used StatGen to generate the BE documentation for Apache APIs that lack documentation by learning from the documentation of the equivalent APIs in JDK. 44% of the generated documentation are rated as useful and 42% as somewhat useful. Second, we used StatGen to detect the inconsistency between the implementations and documentation on behavior exceptions of several JDK8 packages.

In this paper, we make the following contributions:

A. Methodology. A hybrid model for BE documentation ↔ code that works better than the state-of-the-art, rule-based approaches and the deep learning ones, while having the best of both worlds.

B. Applications. We show StatGen's usefulness in 1) generating BE documentation by learning from the equivalent APIs, 2) detecting inconsistency between implementations and BE texts.

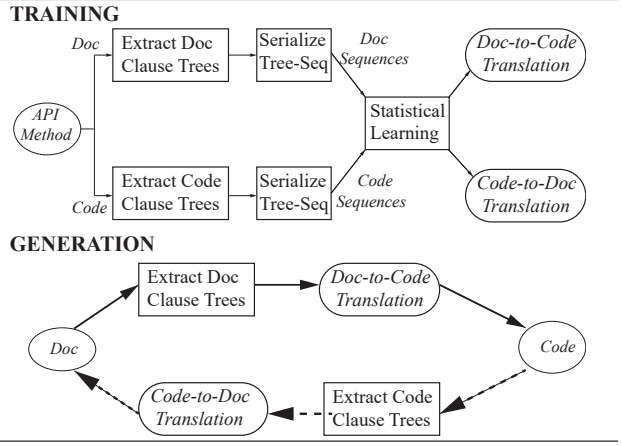
C. Empirical Evaluation. An extensive empirical evaluation instrinsically and extrinsically to show StatGen's higher accuracy than the state-of-the-art approaches. The data is available at [1].

2 KEY IDEAS AND APPROACH OVERVIEW

2.1 Key Ideas

For brevity, we show a simple example of the JDK API substring in Figure 1 (Section 9 shows complex examples from StatGen). The BE documentation is listed before the code: the API throws *StringOutOfBoundsException* if the stated condition occurs. We also extract the corresponding exception-throwing BE code. StatGen aims to translate the BE documentation into the corresponding BE code and vice versa. We design StatGen with the following key ideas:

Figure 2 StatGen: Approach Overview.



2.1.1 **Divide-and-Conquer, Bidirectional Translation.** Feeding the entire BE sentence/code as is to a model would burden it with implicitly learning the structural mappings. In a low-data setting, this is inefficient. Instead, we utilize phrase-based statistical machine translation (SMT) with a structure-based, divide-and-conquer strategy. This is achieved by breaking the given BE sentence/code into smaller fragments, translating individual fragments, and finally merging the results into the final translated BE code/sentences.

2.1.2 **Structural Translation.** Instead of letting the model learn the structure mappings, to enable structural translation, we take advantage of the structure information in BE sentences and BE code by splitting them into the smaller document clause trees and code clause trees. SMT is used to learn and translate the phrases within the clauses in the trees. Such a design choice helps us in avoiding the use of complex models in structure learning.

2.1.3 **Implicit Learning of Translation Rules via Statistical Machine Translation.** While the structure translation is performed deterministically first, SMT is used later to learn the mappings between the words/phrases in a clause and the code tokens/sequences in an expression. SMT is data-driven, helping learn the translation rules implicitly, without having to explicitly specify them, as in the rule-based NLP approaches.

2.2 StatGen: Approach Overview

Figure 2 shows StatGen's overview with two processes. In **training**, we take the corpus of the API methods with BE documentation and source code, parse the documentation to extract the clauses and build the clause trees to represent the conditions for the API stated in the documentation. We also parse the code to extract the exceptional flow graphs and build the code clause trees representing the exception conditions. Clause trees in both sides are serialized into sequences to train the corresponding phrase-based SMT models in two directions. The documentation and code sequences are used accordingly for source and target in each direction.

For **generation** from Doc-to-Code, we parse the BE sentences in Javadoc to extract the clause trees, and convert them into the code clause trees. We then feed the texts in each document clause to the trained SMT model to produce the translated code for each clause. Finally, we merge and serialize the translated code for all the clauses into the final result. A similar process is performed for Code-to-Doc model in the reverse direction.

Figure 3 A Documentation Clause Tree.

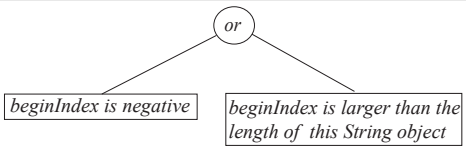
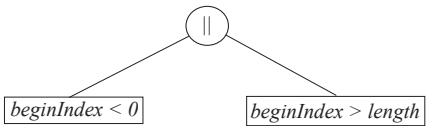


Figure 4 A Code Clause Tree.



3 DOCUMENTATION & CODE CLAUSE TREES

A challenge as using phrase-based SMT for Doc-to-Code is the generated code with incorrect syntaxes [47]. For example, the condition to throw StringIndexOutOfBoundsException, *"if beginIndex is negative or larger than the length of this String object"*, would be translated into $beginIndex < 0 \;||\; > length$. The reason is that the subject beginIndex do not need to be repeated for the part *"larger than the length of this String object"*, however in source code, we need to have $beginIndex > length$. To overcome that, we use tree-structured representations for the texts for the BE conditions and for the source code. For the API subString, the condition of the exception StringIndexOutOfBoundException to be thrown is either *"the beginIndex is negative"* or *"it is greater than the length of this string Object"*. The conditions can be expressed in two forms: texts in documentation (e.g., in English) and expressions in source code (e.g., in Java logical expressions).

3.1 Document Clause Tree

We aim to convert the clause describing a condition (e.g., the condition "if" clause in the documentation) into a tree-structured form, called *a documentation clause tree*. In a clause tree, the root and the intermediate nodes are either the connective words "and" or "or", and the leaf nodes are *simple logical clauses* that cannot be broken further into a conjunctive (via "and") or disjunctive (via "or") of any other clauses. An example of a simple logical clause (simple clause for short) is *"beginIndex is negative"*. Another example is *"beginIndex is larger than the length of this String object"*. A simple clause often has the syntactical structure of either NounPhrase VerbPhrase AdjectivePhrase or NounPhrase VerbPhrase NounPhrase. On the other hand, a clause tree represents a *complex logical clause*, representing a condition with conjunction(s) and/or disjunction(s) among simple clauses and/or other complex clauses. An example of a complex logical clause (complex clause for short) is a clause tree rooted at an "or" node, which has two children (Figure 3). The first child of the root is the simple clause *"beginIndex is negative"* and the second one is the simple clause *"beginIndex is larger than the length of this String object"*. The algorithm to parse documentation into clause trees is in Section 4.1.

Figure 5 A Parse Tree.

```
(ROOT
  (S
    (NP (NN beginIndex))
    (VP (VBZ is)
      (ADJP
        (ADJP
          (JJ negative)
          (CC or)
          (JJR larger
            (PP (IN than)
              (NP
                (NP (DT the) (NN length))
                (PP (IN of)
                  (NP (DT this) (NNP String) (NN object)))))))))))
```

3.2 Code Clause Tree

We parse the code to build a tree for the condition for each exception. The tree is adapted from the structure of a Java logical expression. In this representation, which we refer to as *a code clause tree*, a leaf node is an atomic Java logical expression that cannot be further broken down into a conjunctive or disjunctive of any other logical expressions. The root or an intermediate node is a logical AND (&&) or a logical OR (||) operator (see an example in Figure 4).

4 CLAUSE EXTRACTION ALGORITHMS

4.1 Extracting Documentation Clause Trees

Let us explain how StatGen parses an API documentation to build the clause trees for relevant exceptions. For example, for subString(beginIndex), the pair is StringIndexOutOfBoundException and the clause tree in Figure 3. If the API method is relevant to multiple exceptions, there are multiple pairs of exceptions and clause trees.

StatGen processes the tags throws and exception. For each tag, it extracts the exception's name and the condition clause after the keyword "if" to build a clause tree for that exception. StatGen first identifies the code elements embedded within the documentation such as the method's arguments and fields, and replaces them by special tokens CODE with indexes for their identifications. The rationale for this step is that if they are kept intact, the names of the code elements could be mistakenly treated as regular English terms by the NLP parser to be used for text analysis later. Next, StatGen uses the Stanford NLP tool [25] to parse the text into a parse tree with the part of speech (POS) tagging. In the case of comparative sentences, StatGen heuristically moves prepositional phrases, e.g., *"than the length..."* to be a child of the node of the related comparative adjective e.g., *"greater"* instead of being a child of an adjective phrase as in the NLP parse tree. The parse tree for our running example is shown in Figure 5.

Listing 1 shows the pseudo-code for our clause tree building algorithm. It first decides if the given parse tree *PT* is the case of a conjunction or disjuntion sentence. If it is the case, StatGen identifies the corresponding connective word *CC* at the highest level in the parse tree. The Stanford parser annotates such connective words with the label *CC* (coordinating conjunction). Then, an intermediate node *N* is created in the resulting clause tree corresponding to the operator for *CC* (i.e., either an "and" or "or" node). For each subtree *t* rooted at the *CC* node of the given parse tree *PT*,

Listing 1: Clause Tree Building Algorithm

```
1  function ClauseTree CTBuild (PT: ParseTree)
2    ClauseTree CT
3    if PT is a conjunction/disjunction sentence with a connective word CC
4      OP = convertIntoLogicalOperator(CC)
5      N = new ClauseTreeIntermediateNode (OP)
6      foreach subtree t rooted at CC of PT
7        if t is not a complete English clause, do t = Enhance(t)
8        CT = N ⊕ CTBuild (t)
9    else
10     LeafNode L = new LeafNode(textsOf(PT))
11     CT = new ClauseTree(L)
12   return CT
13
14 function Enhance (t, PT: ParseTree)
15   Identify the missing prefix or suffix of t in PT
16   Duplicate into the subtree p or s of the prefix / suffix
17   Connect t with p or s
18   return t
```

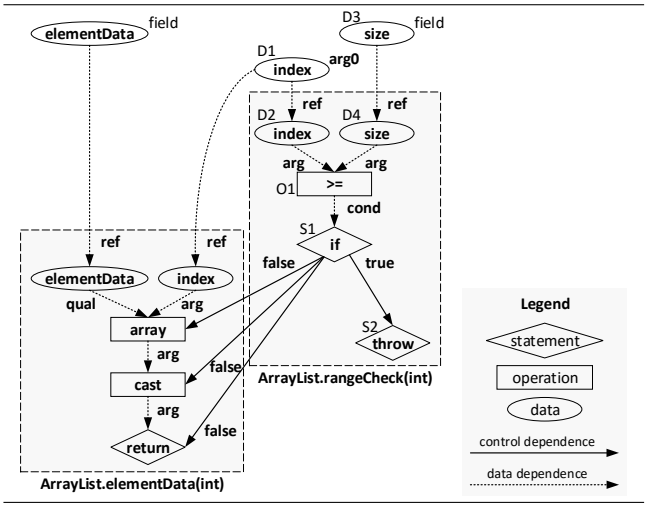
Figure 6 Implementation of java.util.ArrayList.

```
1  public class ArrayList<E> ... {
2      public E get(int index) {
3        rangeCheck(index);
4        return elementData(index);
5      }
6       ...
7      private void rangeCheck(int index) {
8        if (index >= size)
9          throw new IndexOutOfBoundsException(...);
10       }
11      ...
12     E elementData(int index) {
13       return (E) elementData[index];
14     }
15      ...
16     public int size() {
17       return size;
18     }
19 }
```

we check whether *t* is a complete English clause (e.g., in the form of either NounPhrase VerbPhrase AdjectivePhrase or NounPhrase VerbPhrase NounPhrase). If it is not, we need to complete it by calling Enhance(t) (line 7). In our example, the subtree for the phrase *"larger than the length of this String object"* is not a complete English clause.

To complete the phrase with Enhance, StatGen first identifies the missing prefix/suffix using the original parse tree *PT*. For example, the missing prefix for the phrase above is *"beginIndex is"*, which corresponds to the subtrees marked as (NP (NN beginIndex)) and (VP (VBZ is)). StatGen duplicates the subtrees and connects with the incomplete phrase (line 17). For example, after enhancing, we have the parse tree for *"beginIndex is larger than the length of this String object"*. The subtree *t* after being enhanced becomes complete, is used in a recursive call at line 8. Then, the result is formed by connecting the new intermediate node *N* with the resulting clause tree returned from the clause tree built by the recursive call on *t*. At lines 10–11, if the given *PT* is not a conjunction or disjunction, StatGen creates and returns the clause tree with a single leaf node containing the entire texts of *PT*. Finally, the clause tree *CT* is returned.

Figure 7 The PDG of ArrayList.get(int) in Figure 6. Two sub-PDGs in dashed boxes represent two methods called inside its body.



4.2 Extracting Code Clause Trees

Let us explain how we build the clause trees for corresponding code. We first build an intra-procedural PDG (iPDG) for the body of the API. We inline the methods called in the iPDG (Section 4.2.1) and partially evaluate the expressions with concrete values resulting from method inlining (Section 4.2.2). We then extract the parts of PDG that lead to exception exit points to construct the inter-procedural EFG (Section 4.2.3). Finally, we collect the conditions in the EFG to build the code clause trees of logical expressions (Section 4.2.4). Next, let us provide the details of these steps.

4.2.1 Method Inlining. To be complete in detecting exceptional flow paths, we need to perform inter-procedural analysis. The PDG for the body of an API might contain calls to other methods whose bodies contain control conditions and exception exit points. In Figure 6, the method get(index) calls rangeCheck(index) to check the control condition index>=size before throwing an exception. In this work, instead of supporting inter-procedural analysis, we inline the PDGs of those callees to reveal all the conditions and exit points. The inlining process stops when there is no callee or the bodies of the callees are not available, i.e., calling to a method in an external library or calling to a native method. We also stop inlining if encountering a recursive call. Inlining is realized through the connections of the PDG g_m of a method m into the PDG g of its caller.

4.2.2 Partial Evaluation. When inlining g_m in g, the calling context of the method m in its caller could help partially evaluate expressions in g_m. For example, substring(begin) calls substring(begin, end) in its body and passes the field count as the second parameter in the place of the formal parameter end. The exception handling occurs in the body of substring(begin, end). Thus, during inlining, the expression end>count becomes count>count and can be evaluated to false. This partial evaluation helps eliminate infeasible paths and/or simplify the control conditions in detecting the exceptional flows.

4.2.3 Exceptional Flow Graph Extraction. After building intra-procedural PDGs, performing method inlining, and partial evaluation, we extract the EFG. An EFG of an API is a slice of the constructed PDG leading to all exception exit points. We consider all throw and assert statements as exception exit points. We do not consider method invocation nodes because they would or would not lead to an exception. Using conditions along the paths that it is not sure if they lead to exception would result in preconditions stronger than needed.

4.2.4 Condition Abstraction. The preconditions extracted from the implementation could be invisible from outside of the library, e.g., client code, due to information hiding. For example, we could get header.next == header in which header is a private field. However, we aim to derive general preconditions accessible from client code. Therefore, we need to perform the abstraction step on the conditions.

Predicate Method Replacement. We overcome the above problem by searching for a predicate method that is equivalent to the expression containing a private member, and then replace it. For example, the private field size on line 9 of Figure 6 can be replaced with the predicate method size() (lines 17–19).

The search is performed by comparing the PDG of each expression with the PDG of each predicate method in the library (in case that a predicate is a static method in another class). We only search for predicate methods that have no side-effects. The attribute key is used as label when comparing data nodes. We use an approximate method [43] to efficiently compare graphs via vectors.

Argument Replacement. The variables/expressions passed as parameters of an API could be named differently from the names of formal arguments of the API. Thus, we abstract the formal argument names by its position in the argument list. We use the *symbolic name* argi to denote the *i*-th argument in the argument list of the API. For example, the label index of node D1 is abstracted into arg0 because it is the first argument of the API. We use the def-use relation via the ref edges to identify the references of the arguments of the API and abstract their names. For example, for the condition index>=size, operation node O1 with two operands D2 and D4, will be abstracted into arg0>=size because D2 refers to the first argument of the API call.

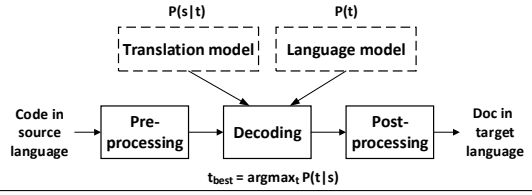
4.2.5 Condition Extraction and Clause Tree Building. We traverse the EFG and collect the guard conditions leading to the exception(s). The conditions are derived by negating these guard condition expressions. A condition is a single clause in the conjunctive normal form of the negated logical expressions. Symbolic execution can be used to better track the changes to the variables in a condition. The clause of logical expressions is parsed into a clause tree (e.g., Figure 4). We discard the clauses having the API's private members.

5 HYBRID TRANSLATION WITH STRUCTURAL & STATISTICAL INFERENCE

5.1 Background on Phrase-based SMT

We leverage a statistical machine translation (SMT) model (Figure 8) in which the parameters of a statistical model are trained from a corpus of source and translation texts [26]. In SMT, translation is the process to produce a sequence t in the target language T from a sequence s in the source language S, where t is the most likely

Figure 8 Statistical Machine Translation [26].



Listing 2: Clause Tree Translation Algorithm

```
1  function DoctoCodeTranslate(Doc D)
2      ClauseTree T = BuildDocClauseTree(D)
3      ClauseTreeTranslate (T)
4      return T.unparseToCode()
5
6  function CodetoDocTranslate(Code C)
7      ClauseTree T = BuildCodeClauseTree(C)
8      ClauseTreeTranslate (T)
9      return T.unparseToDoc()
10
11 function ClauseTreeTranslate (ClauseTree T)
12     for each leaf node L in T
13       L.label = SMT(L.label)
```

Figure 9 An inconsistency between the documentation and implementation of an exception in JDK 1.8.

```
1
2  /** ...
3  * ??param pos the position of the selected item
4  * ??exception IllegalArgumentException if the specified
5  * position is greater than the
6  * number of items or less than zero ... */
7  public synchronized void select(int pos) {
8    if ((pos >= pItems.size()) || (pos < 0)) {
9      throw new IllegalArgumentException("illegal Choice item position :" +
           pos);
10   }
11   if (pItems.size() > 0) {
12     selectedIndex = pos;
13     ChoicePeer peer = (ChoicePeer)this.peer;
14     if (peer != null) {
15       peer.select(pos);
16     } ...
```

sequence according to the translation and language models. The *language model* specifies how likely sequence t occurs in the target language while the *translation model* specifies how likely two sentences s and t co-occur in the corresponding texts of the two languages in the training corpus. Those two models in SMT maintain the probabilities $P(t)$ (probability of the sentence t occurring in the target language) and $P(s|t)$ for all possible sequences s and t (the probability of the mapping between t and s). Those probabilities are estimated from the training corpus of code and translated texts. Once trained, the SMT model can be used for translation.

To apply SMT to our problem, first, for the language model on the target side, we use the n-gram model. For training, the BE documentation and corresponding source code for each exception are parsed to build the clause trees. StatGen then unparses the tree to create the corresponding phrases (n-grams) between texts and code to build the parallel corpus. For Doc-to-Code, the n-gram model is used on the source code, while for Code-to-Doc, it is used on the natural language texts for the BE documentation. The generating probability of a unit in a sequence is dependent only on its *local context*, *i.e.,* a window called n-gram of n previously generated units. Second, for the translation model, we use the phrase-based SMT. More details on phrase-based SMT can be found in [26]. Next, let us explain in details how we use SMT for our problem.

5.2 Hybrid Translation with Clause Trees

This section explains our algorithm to translate an API documentation for a behaviorial exception into corresponding source code. The pseudo-code for translation for each exception tag is displayed in Listing 2. For the Doc-to-Code translation, StatGen first builds the clause trees for the documentation as explained in Section 4.1. It then performs a hybrid translation from each documentation clause tree to a code clause tree ClauseTreeTranslate at line 11. The structure of all the intermediate nodes remains the same and we use the SMT model (Figure 2) to perform the translation for the text of each leaf node. The label of an intermediate node is also converted (e.g. *or* to ||). For example, the doc clause tree in Figure 3 is converted into the code clause tree in Figure 4. StatGen performs Code-to-Doc translation in the same manner. At line 13, the label represents either the text of a leaf node in a documentation clause tree or the expression of a leaft node in a code clause tree. Finally, the translated clause tree of each exception is unparsed and the result is returned.

6 CONSISTENCY CHECKING WITH STATGEN

Let us present an application of StatGen in checking the consistency between BE documentation and source code. In principle, the documentation and implementation of an API method must be consistent (i.e., having the same intent). However, the consistency is not always maintained in practice. For example, Figure 9 shows an inconsistency between the BE documentation and the implementation in the class java.awt.Choice of JDK 1.8. Especially, in this example, the BE documentation (lines 2–5) describes that an exception (IllegalArgumentException) is made if the argument *position*'s value is greater than the number of items or less than 0. This means that an IllegalArgumentException is thrown when the value of parameter pos satisfies the condition (pos > pItems.size()) || (pos < 0). This condition is inconsistent with the actual implementation (lines 7–9), in which the condition to throw the exception is (pos >= pItems.size()) || (pos < 0). The inconsistency between BE documentation and implementation might lead to API misuses and security vulnerabilities [41]. Therefore, there is a need to have an automated tool to detect such inconsistency between BE documentation and implementation.

In general, to check such consistency, our algorithm first transforms the two abstractions to the same representation level. In this work, this has been done by using StatGen to generate source code from the API BE documentation. The consistency between them is formally verified by checking the equivalence of the actual implementation and the generated code.

Listing 3 describes our inconsistency checking algorithm. Given an API method (e.g., the example in Figure 9), it first extracts the documentation and implementations of the behavioral exceptions

Listing 3: Consistency Checking Algorithm

```
1  function CheckConsistency (APIMethod m) {
2    doc = ExtractExceptionDoc (m)
3    impl = ExtractExceptionImpl (m)
4    if (NotReferToSameExpr (doc, impl)) return false
5    Exceptions = PairDocImpl (doc, impl)
6    for all e ∈ Exceptions
7      genImpl = GenerateImplementation (e.doc)
8      actExp = ToLogicExpr (e.impl)
9      genExp = ToLogicExpr (genImpl)
10     if NotEquivalent (actExp, genExp) return false
11   return true
```

(line 2 and line 3). In this example, there is only one exception documentation and one implementation: the text in Javadoc from lines 2–5 and the code fragment from lines 7–9. To check the consistency between documentation and source code, the algorithm inspects the consistency in the set of exceptions referred by the documentation and the code (line 4). Then, for each exception, the algorithm checks the consistency between the precondition described by the documentation and the condition expression implemented by the source code (lines 5–10). Note that before checking the consistency for each exception, in line 5, the algorithm pairs each exception's documentation and its corresponding source code.

In order to check the consistency between the precondition and the actual implemented condition expression for an exception, StatGen generates a condition expression from the precondition in its documentation. In the running example, the generated condition for the case of an IllegalArgumentException being thrown is ((pos > pItems.size()) || (pos < 0)). After that, in order to formally check the consistency, the generated implementation and the actual code are transformed to the logic expressions, $\Phi_{generated}$ and Φ_{actual}. For the example, we have the following

$$\Phi_{generated}\,(pos, pItems\_size) = [(pos > pItems\_size) \ \vee (pos < 0)]$$

$$\Phi_{actual}\,(pos, pItems\_size) = [(pos \geq pItems\_size) \vee (pos < 0)]$$

Finally, to examine the equivalence of $\Phi_{generated}$ and Φ_{actual}, we use the SMT-solver Z3 [13]. For the running example, the first order logic expression that needs to be checked is:

$$\forall\, pos\ \forall pItems\_size\ \Phi_{generated}\,(pos, pItems\_size) \iff$$

$$\forall pos\ \forall pItems\_size\ \Phi_{actual}\,(pos, pItems\_size)$$

The result on the equivalence of $\Phi_{generated}$ and Φ_{actual} from the SMT solver indicates the consistency between the BE documentation and implementation.

7 EMPIRICAL EVALUATION

To evaluate StatGen, we seek to answer the following questions:

RQ1.[Intrinsic Evaluation]: Code-to-Doc. How well does it generate BE documentation compared with the state-of-the-art models?

RQ2.[Intrinsic Evaluation]: Doc-to-Code. How well does it generate code from BE documentation compared with existing models?

RQ3.[Extrinsic Evaluation]. Usefulness. How accurately does StatGen generate behavioral exception documentation for equivalent code in JDK and Apache whose documentation is missing?

RQ4.[Extrinsic Evaluation]. Usefulness. How useful is StatGen's generated behavioral exception documentation in checking the consistency between documentation and source code?

7.1 Experimental Methodology

We followed the five experiment steps described in Wohlin *et al.* [66]: scoping, planning, execution, analysis, and result presentation to conduct the experiments to answer the aforementioned research questions. Let us first start with the data collection.

To train the SMT model, we built a parallel corpus of BE source code-documentation pairs, comprising the method implementations and their JavaDoc documentation from Java Development Kit (JDK, jdk1.8.0_91). In total, we parsed 1,869 JDK classes with 6,802 methods. For the Javadoc of each method, we processed the @throws and @exception descriptions. For each method implementation, we constructed a Program Dependence Graph (PDG). In the PDG, considering all throw statements as exception exit points, we collected all conditions along a slice of the PDG that leads to such exit points. Such a slice of the PDG is referred to as the Exceptional Flow Graph. We follow the exception flow analysis technique in Allen and Horwitz [2]. We then build the document and code clause trees. We unparse the clause trees to form BE code-documentation pairs. The methods without BE specification were discarded. Overall, we have 1,524 pairs: 1,371 pairs (90%) of which are used for training the n-gram and mapping models; and 153 pairs (10%) are used for testing. In our experiments, we use Phrasal SMT [10] (n=7). StatGen is general for any phrase-based SMT. n=7 is the default value in Phrasal. If n >7, the number of phrases increase significantly. If n < 7, the phrases might be too short to capture the clauses.

We chose the following baselines including the rule-based NLP, Statistical Machine Translation, and Deep Learning approaches:

1) Phrase-based SMT in Phan *et al.* [47] (*SMT, sequence, no code structure considered, both translation directions*); (RQ1, RQ2),

2) Dual-task learner [62] (*seq2seq LSTM, dual-task learning, no structure considered, both directions*); (RQ1, RQ2),

3) Hybrid-DeepCom [22] (*neural machine translation with encoder-decoder, attention, code structure considered, code-to-doc*) (RQ1),

4) TreeGen, a tree-based transformer [56] (*AST structure, attention mechanism of Transformer, doc-to-code*) (RQ2),

5) Jdoctor [8], a rule-based, Doc-to-Code approach (pre-defined rules for lexical and semantic matching of texts) (RQ2),

6) Zhou *et al.* [71], a rule-based model, is used for RQ4 (consistency checking). It does not explicitly generate BE code, thus, we do not compare it in RQ2. The tools in 1–5 do not check consistency.

7.2 Infer BE Documentation from Code (RQ1)

For the test set constructed with the randomly-selected BE code-documentation pairs, we compared the translated sequences (*results*) with the corresponding Javadoc sequences (*references*). We compute the longest common subsequence (LCS) to assess the similarity between the result and reference sequences while considering their token order. Utilizing LCS, we report two evaluation metrics, Precision and Recall, which are formulated as follows: $Precision = \frac{|LCS|}{|Result|}$, $Recall = \frac{|LCS|}{|Reference|}$. They are accumulatively computed for all the sequences in the testing pairs. High

Table 1: Code-to-Documentation Inference Result.

	L_G	L_R	Pre.	Rec.	BLEU	Match	Same	Close	Incor
Phan *et al.* SMT [47]	12	8	58%	79%	43%	27%	12%	25%	36%
Dual-Task Learner [62]	12	8	61%	70%	30%	23%	13%	10%	54%
Hybrid-DeepCom [22]	12	8	62%	73%	35%	25%	15%	11%	49%
StatGen	10	8	75%	81%	56%	41%	7%	15%	37%

Figure 10 Examples of "Same"/"Close" Generated Documentation.

Generated: if horizon be **less than 0** throw IllegalArgumentException
Reference: if horizon be **negative** throw IllegalArgumentException

Generated: if **key or** value be null throw NullPointerException
Reference: if value be null throw NullPointerException

Recall means that the generated sequences cover more words in the references in the right order. High *Precision* means that more words in the generated sequences are in the correct order.

We also measured BLEU score (0–1), a popular NLP metric measuring translation accuracy of *all possible phrases with various lengths*. Specifically, $\text{BLEU} = BP.e^{\frac{1}{n}(\log P_1 + \ldots + \log P_n)}$. P_i is the metric measuring the overlapping between the bag of i-grams appearing in the results and that of i-grams appearing in the references. BP equals to the ratio between the two lengths. We also manually compared the semantics of the generated documentation with the references with four categories: (1) the ones that are textually **matched** with the references; (2) the ones that are not exactly matched but have the **same** semantics; (3) the ones that are semantically **close** with the reference and need slight modification for correction; and (4) the ones that are **incorrect** or have totally different semantics.

As seen in Table 1, StatGen achieves precision of 75%, recall of 81% and BLEU score of 56% in Code-to-Doc. The lengths of the generated documentation sequences (L_G) are close to those of the references (L_R). In terms of semantic accuracy, 48% of generated documentations are correct, 41% of which are exactly matched with the expected. In 7% cases, the generated documentation contains phrases having the same semantics as in references even though they are not lexically matched. For example, we generated the phrase "*less than 0*" while the text in the reference is "*negative*" as shown in Figure 10. There are 15% of generated sentences that are semantically close to the references. For those cases, deleting or adding a few words, or renaming an identifier to match with an argument name of the API would make them correct (Figure 10).

Compared with Phan *et al.* [47], StatGen improves relatively across all evaluation metrics: 29.3% in Precision, 2.5% in Recall, and 30.2% in BLEU. These improvements can be attributed to StatGen's design choice of leveraging structures in both documentation sentences and code, which is not the case with Phan *et al.* [47].

Compared with the Dual-task learning model [62], StatGen observes also high relative improvements: 22.9%, 15.7%, and 86.6% in Precision, Recall, and BLEU score. Upon examining the results, we see that similar to the results from Phan *et al.* [47], many generated documentation sentences are grammatically incorrect and meaningless. Despite supporting forward and back-translation, both baselines (Phan *et al.* [47] and Dual-task learning [62]) with their sequence representations are inadequate in learning the structure in sentences and source code, and consequently, the mappings between them. In contrast, StatGen parses both texts and source code, and maps the structures. This reduces the cascading effect of incorrect learning and mapping of structures.

Compared with Hybrid-DeepCom [22], StatGen relatively improves 20.9%, 10.9%, and 60% in Precision, Recall, and BLEU score, respectively. Unlike Phan *et al.* [47] and Dual-task learning [62], Hybrid-DeepCom [22] does consider structure, but only on the source code side, and not for the document sentences. This is one of the potential reasons for the lack in performance.

Importantly, the lack of training data (only 1,371 pairs) is a factor making those deep learning-based models (i.e., the dual-task learner [62] and tree-based neural machine translation in Hybrid-DeepCom [22]) less accurate than StatGen. StatGen makes explicit the translations of structures in documentation and code, and leverages the phrase-based SMT model to learn the mappings of the short phrases. Thus, with the divide-and-conquer strategy, SMT-based StatGen does not need as much data as those DL models.

The attention mechanism in Hybrid-DeepCom [22] (Table 1), also would not improve much due to the lack of training data.

Moreover, except Phan *et al.* [47], all the above DL models serve the general purpose of code-to-text translation, rather than targeting and exploring the structures in BE documentation and source code. For this problem, StatGen learns to build the document clause trees in BE documentation and code structures and map them to reduce the incorrect mappings between the structures in two sides.

7.2.1 Ability to learn the mappings between words and code tokens. To study the knowledge that StatGen learned, we examined the mapping table produced as the by-product of model training. We focused on the mappings of operators and methods' argument names that are used in the conditions of exception behaviors. First, we examined 9 operators, including conditional, unary, equality, and relational operators in Java. We manually checked the corresponding mapping phrases for those operators in the mapping table produced by Phrasal SMT. If a phrase is a correct description of an operator, we note that as a correct one in the resulting list. We found that StatGen produced the *correct descriptions of 8 out of 9 operators* at the top rank, which gives the top-1 accuracy of 89%. Top-2 accuracy and top-3 accuracy are also 89%. All correct results of these operators are in top-4 mappings.

Second, we randomly examined 100 identifiers used as the argument names of API calls and were used in the conditions leading to exceptions. StatGen achieves *76% top-1 accuracy*, i.e., in 3 out of 4 cases, it maps the identifiers in the code to the correct names in Javadoc at the top rank. It maps almost all the names correctly at top-3 positions and maps them all correctly at top-7 positions.

7.3 Infer Code from BE Documentation (RQ2)

We trained StatGen in the Doc-to-Code direction. With an independent test set of 153 randomly-selected documentation-code pairs as the input of the trained model, we generate the corresponding code sequences and compare against the references. As

Table 2: Documentation-to-Code Inference Result.

	L_G	L_R	Pre.	Rec.	BLEU	Match	Same	Close	Incor
Phan *et al.* [47]	18	17	76%	77%	57%	29%	2%	18%	52%
Jdoctor [8]	21	17	68%	65%	48%	19%	4%	16%	61%
Dual-Task Learner [62]	23	17	60%	59%	26%	11%	3%	10%	76%
TreeGen [56]	22	17	61%	69%	42%	18%	5%	18%	59%
StatGen	18	17	75%	84%	63%	47%	4%	12%	37%

in Section 7.2, we use the same metrics: Precision, Recall, and BLEU score. $Precision = \frac{|LCS|}{|Result|}$, $Recall = \frac{|LCS|}{|Reference|}$. The generated code is treated as the sequence of code tokens and LCS is computed on the sequences. The rationale is that those metrics are applicable to all compared models. Any structure-aware metric will not work for the baselines that do not consider code structures.

As seen in Table 2, StatGen achieves high accuracy of 75% precision, 84% recall, and 63% BLEU score. For semantic accuracy, 51% generated code fragments are correct, *47% of which are exactly matched with the references* and 4% others have the same semantics as expected. Such an example is shown below: StatGen generates the sequence `!(size > 0)` while the reference is `size <= 0`.

```
Generated: if ( ! ( size > 0 ) ) throw new IllegalArgumentException ( ) ;          (1)
Reference: if ( size <= 0 ) throw new IllegalArgumentException ( ) ;
```

There are 12% of generated code fragments that have semantics close to the references. All of them have syntax errors. Deleting or adding a punctuation or a few tokens, or renaming an identifier to match with an API's argument name would make them correct. In the example below, one needs to rename `index` to `fromIndex` and replace the redundant tokens at the end with a semi-colon.

```
Generated: if ( index < 0 ) throw new IndexOutOfBoundsException ( ) != null ) if ( (2)
Reference: if ( fromIndex < 0 ) throw new IndexOutOfBoundsException ( ) ;
```

Compared with Phan *et al.* [47], StatGen relatively improves 9.1% and 10.5%, in Recall and BLEU. Examining the result, more generated code is syntactically incorrect for Phan *et al.* [47] since it treats source code as sequences.

Compared with Jdoctor [8], StatGen improves relatively 10.3%, 29.2%, and 31.2% in Precision, Recall, and BLEU score. We examine the results and found that the inaccuracy with Jdoctor is due to 1) its specificity of the lexical/semantic patterns of texts, 2) the lack of rules in handling complex cases, e.g., "if the length of the subtring is smaller than or equal to the length of this string". StatGen's document clause parser handles well such complex cases.

Compared with Dual-Task Learner [62], StatGen improves relatively 25%, 42.4%, and 142% in Precision, Recall, and BLUE score.

Compared with TreeGen [56], a tree-based transformer, StatGen also observes high relative improvements: 22.9%, 21.7%, and 50% in Precision, Recall, and BLUE score.

The Dual-Task Learner [62] treats both code and documentation as sequences of tokens. In contrast, TreeGen [56] considers code structure, but not the structure of the sentences in BE documentation. Thus, both models are burdened with implicitly learning of the text and code structures and the mappings between them. This further raises the need for more training data, which is not available for this BE documentation. In contrast, StatGen, which considers structure for both code and documentation, and uses a significantly less complex ML algorithm, is more suitable and effective.

Table 3: User Study Result.

	Correct	Good Start	Not Sure	SWIncor	Incorrect	Total
Number	48	45	4	6	5	108
Percentage	44%	42%	4%	6%	4%	100%

8 GENERATING DOCUMENTATION FOR UNDOCUMENTED API METHODS (RQ3)

This section presents an experiment to show StatGen's usefulness in documentation generation. We used StatGen to generate the behavior exception documentation for Apache APIs that lack documentation by learning from the BE documentation of the equivalent APIs in JDK. We first trained Code-to-Doc model with all 1,524 pairs of JDK API methods, and derived the BE documentation for all API methods in `apache.commons.collections` that do not have pre-condition documentation. Since they do not have documentation, we cannot use Precision, Recall, or BLEU score. Thus, we used human judgement on the BE documentation generated by StatGen. We recruited three human subjects with several year experience on specification, JDK library, and Javadoc to evaluate the result.

For preparation, each human subject was shown an example of an JDK method (Figure 6) along with the documentations that StatGen created for that API. We then pre-selected the correct answer (based on the ground-truth) and explained why it is "correct", "a good starting point", "not sure", "somewhat incorrect", and "incorrect". "Correct" means that the documentation can be used as-is (Figure 1). "Good starting point" means that it needs small changes to be used in a documentation, such as changing a comparison operator from strict to non-strict (Example (1), Section 7.3). "Incorrect" means that the condition is totally irrelevant in writing the documentation. "Somewhat incorrect" means that the generated documentation requires more changes for correction (Example (2), Section 7.3).

Next, each subject was shown a method and the generated documentation. (S)he was asked to rate the result and also given an opportunity to write general comment. In total, (s)he rated 108 randomly selected documentation for 108 API methods in Apache. For the cases that the ratings from three human subjects were different, they had a discussion to reach a consensus.

As seen in Table 3, the human subjects rated 44% of the results as correct. While other results are not entirely correct, 42% of them are deemed to be a good starting point for the human subject to begin writing the BE documentation. Excluding the not-sure responses, only 10% of the results were deemed to be incorrect and somewhat incorrect. Only 4% are considered to be totally irrelevant. In other words, 86% of the results are considered as correct and useful.

Examples. Here are some useful and somewhat-useful examples:

```
(1) IndexOutOfBoundsException ( index is less than 0 or index is greater than size() )
(2) NoSuchElementException ( there be no setNextObject() )
```

Table 4: Inconsistency Checking for Packages in javax.

package	SQL	XML	Lang	Sec	Util	Management	Others	Total
pairs	150	178	122	180	520	178	2,228	3,557

The first one is inferred for getValue(int) in LinkedMap of org.apache.commons.collections.map. The human subject rated it as correct since it reflects correctly the conditions in the code. The second example is inferred for next() of the FilterIterator class from org.apache.commons.collections.iterators. The inferred documentation states that NoSuchElementException will be thrown if there is no setNextObject(), whereas, the expected condition for the API is that if nextObjectSet returns false and not followed by an invocation to setNextObject(), NoSuchElementException is thrown. As the inferred documentation is partially correct and close to what is expected, the human subjects rated it as somewhat correct.

9 CONSISTENCY CHECKING (RQ4)

This section presents another StatGen application in detecting the inconsistencies between BE documentation and implementations of several packages in JDK8. We compared it against the state-of-the-art approach by Zhou *et al* [71], which follows rule-based NLP. We used the same testing dataset and ground truth used in their work. The dataset includes the code and Javadoc documentation on the core packages in JDK. We ran StatGen with the detection algorithm in Listing 3 on the dataset. Because Zhou *et al.* [71]'s technique works for general documentation, we only compared the tools on BE documentation and source code. Specifically, we compared the consistency checking only for the pairs of @exception or @throw TagElement in documentation and the corresponding throw statements in the source code that handle the same exceptions. We also considered the cases where the documentation of an API is inherited from the parent class. Thus, if a tool sees the Javadoc tag @inheritDoc, it will consider the description in the parent class.

We use all the pairs of exception conditions between documentations and implementations extracted from API methods in JDK8 core packages (under java) as the parallel corpus to train StatGen. Finally, we have the dataset (Table 4) containing 3,557 methods that have both Javadoc documentations describing exception behaviors and implementation code throwing the matched exceptions.

Because we evaluate StatGen in consistency checking and for comparison, we use the same metrics as in Zhou *et al.* [71] (rather than the metrics in RQ1). Thus, Precision_Det is defined as the ratio between the number of cases that a tool correctly detected as inconsistency over the total number of detected cases. Recall_Det is computed as the ratio between the number of cases that a tool correctly detected as inconsistency over the total number of inconsistent cases. F-score is the harmonic mean of the two metrics: F-score = $2 \times Precision\_Det \times Recall\_Det / (Precision\_Det + Recall\_Det)$.

9.1 Results for All Packages

As seen in Table 5, StatGen improves with 27% more correct and 28% more complete result than the rule-based model Zhou *et al.* [71].

9.1.1 Correct Cases. As the first example, with the exception IllegalArgumentException thrown in the constructor LoggingPermission(String,String) in package java.util.logging, the documentation is *"name empty or argument invalid"*. StatGen produces the code (name.equals("") || actions != null). The implementation is inconsistent with this formula.

Table 5: Consistency Checking Result.

	Precision_Det	Recall_Det	F-score
Zhou *et al.* [71]	36%	62%	46%
StatGen	63%	90%	74%

Table 6: Result for each Package in javax.

package	SQL	XML	Lang	Sec	Util	Manage	Total
Precision_Det	88%	84%	82%	61%	65%	64%	**71%**
Recall_Det	93%	81%	78%	81%	85%	75%	**82%**
F-score	90%	83%	80%	70%	75%	69%	**76%**

For the API insert(int,String) of the class java.lang.AbstractStringBuilder, the documentation states the condition for exception throwing is *"the offset invalid"*. For this case, StatGen produced (offset < 0 || offset > length()), which is consistent with the implementation. However, Zhou *et al.* [71] converts the phrase into the predicate UnRecognized(offset) using heuristics on phrase matching. Eventually, their tool decided on an inconsistency, leading to a false positive case.

For the API RoleUnresolvedList(List) in the package javax.management, the exception condition of IllegalArgumentException is described as *"list parameter null or list parameter contains any non-RoleUnresolved object"*. However, in the implementation, the condition is only list == null. StatGen is able to detect and report that inconsistency.

9.1.2 Incorrect Cases. We also investigated incorrect/incomplete cases. An example of incomplete documentation is from the API java.lang.Throwable.setStackTrace(StackTraceElement[]). The condition defensiveCopy[i] == null, which causes NullPointerException, is not described in the documentation. That is, the documentation missed the description on *"stackTrace is null or any element of stackTrace is null"*.

We additionally studied the false positive cases. We found that the documentations in those cases are too abstract and vague. For example, the documentations for some APIs state a condition as *"null parameter"*. The translated result is often not correct. To overcome this, one could encode such case in a rule for post processing in order for the model to produce a better translation by updating the correct parameter in the translated results.

One common source of false negative cases is out-of-vocabulary units, in which StatGen cannot find the translation, mainly because it has not encountered the texts in the input documentation. Therefore, it cannot determine the inconsistency. An example is the case of the API base64toInt(char) of the class javax.management.remote.rmi.RMIConnector. The documentation is unique in our dataset. In such cases, training with a larger data can help StatGen observe similar ones.

Note that while Zhou *et al.* [71] is capable of handling several types of API document directives, for comparison with StatGen, we ran it only on exception handling. That is the reason for the differences between our reported results (Table 5) and the ones in their paper (in which their model was run on all types of directives).

Table 7: Accuracy for Different Condition Types.

Type	Precision_Det	Recall_Det	F-score
NullnessNotAllow	67%	79%	72%
NullnessAllow	80%	100%	89%
RangeLimit	68%	93%	78%
TypeRestrict	60%	100%	75%
OtherType	75%	80%	78%

9.2 Results for Individual JDK Packages

We further analyze StatGen's performance on each JDK package (Table 6). The accuracy for javax.management is lower than others because the documentation contains too abstract sentences such as *"null parameter"*. In contrast, StatGen performs well on javax.sql when the texts in documentation are quite standard English. StatGen was able to use its parsing to produce correct clause trees, leading to better inconsistency detection.

We also analyzed the result according to different condition types for exceptions. Zhou *et al.* [71] classify the documentation/specification into different types that can cause exceptions. For example, NullnessNotAllow and NullnessAllow refer to the conditions in which the parameters of the API method is allowed to be null or not. RangeLimit refers to the conditions on the range limits on the values of the parameters. TypeRestrict refers to the restrictions on the types of the parameters of an API. Zhou *et al.* [71] used different heuristics to detect each type. In contrast, StatGen follows a NLP process to generally parse and translate the documentation into code without pre-defined heuristics for each type of condition. As seen in Table 7, StatGen's accuracy for all condition types are quite stable from 72-89% F-score. In other words, StatGen is quite effective across all different condition types for exception behaviors.

9.3 Threats to Validity

Our experiments were run on Javadoc, however, our method is general. In future, we will evaluate on other documentation formats and programming languages. Our experiments were only on two libraries. We will evaluate on other libraries. The selection of 10% test set might affect the generalization of our results. Our RQ3 experiment was with three human subjects. Despite strong indications from the subjects via the high usefulness ratings, a larger-scale survey would further help in establishing StatGen.

10 LESSONS LEARNED

In this section, we present our lessons learned from building StatGen. NLP tools such as phrase-based SMT and Stanford parsers are well-equipped to handle textual documents. However, *they fall short while applying them directly to software documentation and source code.* First, software documentation has code elements embedded within the texts. Such embedded code elements need to be handled separately before feeding them into an SMT model because they refer to the parameters in the APIs. Second, condition expressions in source code have structures. Representing code as regular texts, phrase-based SMT did not work well as we showed via StatGen's improvement in performance over *Phan et al.* [47]. Third, while a translation model for texts is sufficient with phrase-to-phrase mappings, for source code and software documentation, more customization is required. Specifically, phrase-based translation works better in the divide-and-conquer fashion where it applies to substructures and the translated pieces are merged accordingly into the structure. The techniques used in StatGen provide good lessons for such customizations of SMT for text ↔ code problems. Finally, in future work, the customized SMT model is desirable to handle the direct translation from text structure to code structure.

11 RELATED WORK

StatGen is closely related to the work by Phan *et al.* [47]. In comparison, Phan *et al.* [47] do not consider the internal structure of API documentation and that of source code. The dual-task learner [62] also works in both directions between texts and code, however, it uses only sequences. TreeGen [56], a tree-based transformer, leverage code structure for code generation but does not use the clause structures in documentation. Sun *et al.* [55] generate code by integrating grammar rules of programming languages, but not for texts. Similarly, Hybrid-DeepCom [22] does not leverage sentence structures in generating code comments. Those text-to-code and code-to-text DL models [19, 20, 22, 23, 38, 49, 55, 62] require large training data and are not designed for BE documentation. Moreover, they are not effective in a low-data setting as with this problem.

Jdoctor [8] and its earlier work, Toradocu [18], are rule-based NLP approaches. Toradocu works for exception conditions, while Jdoctor supports more conditions. Jdoctor does not support Code-to-Doc. The nature of SMT enables StatGen work in both directions.

In Zhou *et al.* [71], Javadoc is analyzed and compared against the source code to detect inconsistencies. In comparison, their approach uses heuristics on the patterns of sentences in API documentation to classify them into pre-defined categories such as NullAllow, RangeLimit, etc. StatGen uses a generative approach in SMT without heuristics and pre-defined categories. Tan *et al.* [57] use NLP to analyze API Javadoc to produce test cases to detect inconsistencies. While their approach and ours use NLP to parse the documentation, we do not use NLP rules and patterns as in their work.

C2S [69] translates natural language comments into formal program specifications. It constructs the alignments between natural language word and specification tokens from existing comments and their specifications via NLP rules. StatGen performs structure mappings from both sides and use SMT for clause translation. Mahmud *et al.* [39] performed a comparative study on three source code summarization models. They provided insights on the relations between metric-based performance and model prediction errors.

The approaches on deriving API specifications from source code come from two areas: program analysis and mining software repositories (MSR). Several static analysis approaches [9, 14, 27, 30, 50, 53, 63, 68] analyze the code statically to derive the conditions, however, suffers the high false positive results due to the overestimation of static analysis. Specification mining approaches relying on dynamic analysis [3–7, 15, 17, 28, 29, 33–37, 40, 46, 51, 52, 64] are often incomplete due to the incomplete test cases to reveal the conditions. MSR techniques use the principle of frequently used patterns as correct conditions. However, they focus on usage orders and temporal orders of APIs [16, 31, 32, 44, 48, 58, 60, 61, 65, 67, 70]. SMT has also been used in software engineering problems [21, 24, 45, 59].

12 CONCLUSION

In conclusion, we present a novel hybrid direction of statistical+structural machine translation to support the inference between BE documentation and implementations. Our empirical results showed that StatGen achieves high accuracy in inferring BE documentation from source code and vice versa. While our currently used phrase-based SMT achieves high results, more customization on the mapping and language models are needed to accommodate more formal specifications and source code. A combination of a model for formal languages and a statistical language model is a good direction. With respect to mapping algorithms, a structure-based mapping model, e.g., tree-to-tree could improve the performance.

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation (NSF) grants CNS-2120386, CCF-1723215, CCF-1723432, TWC-1723198, and the US National Security Agency (NSA) grant NCAE-C-002-2021 on Cybersecurity Research Innovation.

REFERENCES

[1] 2022. *Statgen.* https://nguyenhoan.github.io/statgen/

[2] Matthew Allen and Susan Horwitz. 2003. Slicing Java Programs That Throw and Catch Exceptions. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '03)*. Association for Computing Machinery, 44–54. https://doi.org/10.1145/966049.777394

[3] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining Specifications. In *Proceedings of the 29th ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL '02)*. ACM, 4–16. https://doi.org/10.1145/503272.503275

[4] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. 2013. Unifying FSM-inference Algorithms Through Declarative Specification. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, 252–261.

[5] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. 2015. Using Declarative Specification to Improve the Understanding, Extensibility, and Comparison of Model-Inference Algorithms. *IEEE Transactions on Software Engineering* 41, 4 (April 2015), 408–428.

[6] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. 2014. Inferring Models of Concurrent Systems from Logs of Their Behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, 468–479.

[7] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. 2011. Leveraging Existing Instrumentation to Automatically Infer Invariant-constrained Models. In *Proceedings of the 19th Symposium on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. ACM, 267–277. https://doi.org/10.1145/2025113.2025151

[8] 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 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. ACM, 242–253.

[9] Raymond P.L. Buse and Westley R. Weimer. 2010. Automatically Documenting Program Changes. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*. ACM, 33–42.

[10] Daniel Cer, Michel Galley, Daniel Jurafsky, and Christopher D. Manning. 2010. Phrasal: A Statistical Machine Translation Toolkit for Exploring New Model Features. In *Proceedings of the NAACL HLT 2010 Demonstration Session*. Association for Computational Linguistics, Los Angeles, California, 9–12.

[11] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-Tree Neural Networks for Program Translation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*. Curran Associates Inc., 2552–2562.

[12] Barthélémy Dagenais and Martin P. Robillard. 2010. Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, 127–136.

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[14] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada) *(SOSP'01)*. ACM, 57–72. https://doi.org/10.1145/502034.502041

[15] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 21st International Conference on Software Engineering* (Los Angeles, California, USA) *(ICSE'99)*. ACM, 213–224. https://doi.org/10.1145/302405.302467

[16] Mark Gabel and Zhendong Su. 2008. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Atlanta, Georgia) *(SIGSOFT '08/FSE-16)*. ACM, 339–349. https://doi.org/10.1145/1453101.1453150

[17] Carlo Ghezzi, Mauro Pezzè, Michele Sama, and Giordano Tamburrelli. 2014. Mining Behavior Models from User-intensive Web Applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, 277–287.

[18] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic Generation of Oracles for Exceptional Behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, 213–224.

[19] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016 (To appear). Deep API Learning. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM.

[20] Tihomir Gvero and Viktor Kuncak. 2015. Synthesizing Java Expressions from Free-form Queries. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. ACM, New York, NY, USA, 416–432. https://doi.org/10.1145/2814270.2814295

[21] Hideaki Hata, Emad Shihab, and Graham Neubig. 2018. Learning to Generate Corrective Patches using Neural Machine Translation. *CoRR* abs/1812.07170 (2018). arXiv:1812.07170 http://arxiv.org/abs/1812.07170

[22] Xing Hu, Ge Li, Xin Xia, D. Lo, and Zhi Jin. 2019. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* 25 (2019), 2179–2217.

[23] Srini Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *ACL*.

[24] Alan Jaffe, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, and Bogdan Vasilescu. 2018. Meaningful Variable Names for Decompiled Code: A Machine Translation Approach. In *Proceedings of the 26th Conference on Program Comprehension (ICPC '18)*. ACM, 20–30.

[25] Dan Klein and Christopher D. Manning. 2003. Accurate Unlexicalized Parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1 (ACL '03)*. Association for Computational Linguistics, 423–430.

[26] Philipp Koehn. 2010. *Statistical Machine Translation* (1st ed.). Cambridge University Press, New York, NY, USA.

[27] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. 2006. From uncertainty to belief: inferring the specification within. In *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI '06)*. USENIX Association, 161–176.

[28] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. Automatic Mining of Specifications from Invocation Traces and Method Invariants. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, 178–189.

[29] T. B. Le, X. D. Le, D. Lo, and I. Beschastnikh. 2015. Synergizing Specification Miners through Model Fissions and Fusions (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 115–125.

[30] Tao Lei, Fan Long, Regina Barzilay, and Martin C. Rinard. 2013. From Natural Language Specifications to Program Input Parsers. In *ACL*.

[31] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *Proceedings of the 13th Symposium on Foundations of Software Engineering* (Lisbon, Portugal) *(ESEC/FSE-13)*. ACM, 306–315. https://doi.org/10.1145/1081706.1081755

[32] Benjamin Livshits and Thomas Zimmermann. 2005. DynaMine: finding common error patterns by mining software revision histories. *SIGSOFT Softw. Eng. Notes* 30, 5 (2005), 296–305.

[33] D. Lo and S. Khoo. 2006. QUARK: Empirical Assessment of Automaton-based Specification Miners. In *2006 13th Working Conference on Reverse Engineering*. 51–60.

[34] David Lo and Siau-Cheng Khoo. 2006. SMArTIC: Towards Building an Accurate, Robust and Scalable Specification Miner. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*. ACM, 265–275.

[35] David Lo and Shahar Maoz. 2010. Scenario-based and Value-based Specification Mining: Better Together. In *Proceedings of the IEEE/ACM International Conference*

on Automated Software Engineering (ASE '10). ACM, 387–396.

[36] David Lo, Leonardo Mariani, and Mauro Pezzè. 2009. Automatic Steering of Behavioral Model Inference. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*. ACM, 345–354.

[37] David Lo, Leonardo Mariani, and Mauro Santoro. 2012. Learning Extended FSA from Software: An Empirical Assessment. *J. Syst. Softw.* 85, 9 (Sept. 2012), 2063–2076.

[38] Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A Neural Architecture for Generating Natural Language Descriptions from Source Code Changes. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 2: Short Papers*, Regina Barzilay and Min-Yen Kan (Eds.). Association for Computational Linguistics, 287–292. https://doi.org/10.18653/v1/P17-2045

[39] Junayed Mahmud, Fahim Faisal, Raihan Islam Arnob, Antonios Anastasopoulos, and Kevin Moran. 2021. Code to Comment Translation: A Comparative Study on Model Effectiveness & Errors. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*. Association for Computational Linguistics, Online, 1–16. https://doi.org/10.18653/v1/2021.nlp4prog-1.1

[40] Leonardo Mariani and Fabrizio Pastore. 2008. Automated Identification of Failure Causes in System Logs. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering (ISSRE '08)*. IEEE Computer Society, 117–126. https://doi.org/10.1109/ISSRE.2008.48

[41] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping Through Hoops: Why Do Java Developers Struggle with Cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 935–946. https://doi.org/10.1145/2884781.2884790

[42] Hoan Anh Nguyen, Robert Dyer, Tien N. Nguyen, and Hridesh Rajan. 2014. Mining Preconditions of APIs in Large-scale Code Corpus. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, 166–177.

[43] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. 2009. Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection. In *FASE '09*. Springer Verlag, 440–455.

[44] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-based Mining of Multiple Object Usage Patterns. In *Proceedings of Conference on the Foundations of Software Engineering (ESEC/FSE '09)*. ACM, 383–392.

[45] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation (T). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15)*. IEEE Computer Society, 574–584.

[46] Tony Ohmann, Michael Herzberg, Sebastian Fiss, Armand Halbert, Marc Palyart, Ivan Beschastnikh, and Yuriy Brun. 2014. Behavioral Resource-aware Model Inference. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, 19–30.

[47] H. Phan, H. A. Nguyen, T. N. Nguyen, and H. Rajan. 2017. Statistical Learning for Inference between Implementations and Documentation. In *2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER)*. 27–30. https://doi.org/10.1109/ICSE-NIER.2017.9

[48] Michael Pradel and Thomas R. Gross. 2009. Automatic Generation of Object Usage Specifications from Large Method Traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, 371–382. https://doi.org/10.1109/ASE.2009.60

[49] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: Synthesizing What I Mean. In *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*. ACM Press.

[50] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Static Specification Inference Using Predicate Mining. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. ACM, 123–134. https://doi.org/10.1145/1250734.1250749

[51] S. P. Reiss and M. Renieris. 2001. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*. IEEE CS, 221–230.

[52] Matthias Schur, Andreas Roth, and Andreas Zeller. 2013. Mining Behavior Models from Enterprise Web Applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, 422–432.

[53] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. 2010. Towards Automatically Generating Summary Comments for Java Methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*. ACM, 43–52.

[54] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API Documentation. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. 643–652.

[55] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A Grammar-Based Structural CNN Decoder for Code Generation. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence* (Honolulu, Hawaii, USA) *(AAAI'19/IAAI'19/EAAI'19)*. AAAI Press, Article 866, 8 pages. https://doi.org/10.1609/aaai.v33i01.33017055

[56] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. TreeGen: A Tree-Based Transformer Architecture for Code Generation. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 8984–8991. https://ojs.aaai.org/index.php/AAAI/article/view/6430

[57] 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 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST '12)*. IEEE CS, 260–269.

[58] Suresh Thummalapenta and Tao Xie. 2009. Alattin: Mining Alternative Patterns for Detecting Neglected Conditions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, 283–294. https://doi.org/10.1109/ASE.2009.72

[59] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On Learning Meaningful Code Changes via Neural Machine Translation. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, 25–36.

[60] Andrzej Wasylkowski and Andreas Zeller. 2009. Mining Temporal Specifications from Object Usage. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, 295–306. https://doi.org/10.1109/ASE.2009.30

[61] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting Object Usage Anomalies. In *Proceedings of the Symposium on Foundations of Software Engineering* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. ACM, 35–44. https://doi.org/10.1145/1287624.1287632

[62] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. *Code Generation as a Dual Task of Code Summarization*. Curran Associates Inc., Red Hook, NY, USA.

[63] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. 2011. Inferring Better Contracts. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. ACM, 191–200. https://doi.org/10.1145/1985793.1985820

[64] Westley Weimer and George C. Necula. 2005. Mining Temporal Specifications for Error Detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Edinburgh, UK) *(TACAS'05)*. Springer-Verlag, 461–476. https://doi.org/10.1007/978-3-540-31980-1_30

[65] Chadd C. Williams and Jeffrey K. Hollingsworth. 2005. Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques. *IEEE Trans. Softw. Eng.* 31, 6 (2005), 466–480.

[66] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.

[67] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. ACM, 282–291. https://doi.org/10.1145/1134285.1134325

[68] Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao, and Feng Qin. 2016. Automatic Model Generation from Documentation for Java API Functions. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, 380–391.

[69] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. 2020. C2S: Translating Natural Language Comments to Formal Program Specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 25–37. https://doi.org/10.1145/3368089.3409716

[70] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and Recommending API Usage Patterns. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*. Springer-Verlag, 318–343.

[71] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs Documentation and Code to Detect Directive Defects. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, 27–37.