Documentation-Based Functional Constraint Generation for Library Methods

Renhe Jiang[1], Zhenzhao Chen[1], Yu Pei[2], Minxue Pan[1*], Tian Zhang[1*], Xuandong Li[1]

[1] *State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China*
[2] *Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China*

SUMMARY

Although software libraries promote code reuse and facilitate software development, they increase the complexity of program analysis tasks. To effectively analyze programs built on top of software libraries, it is essential to have specifications for the library methods that can be easily processed by analysis tools. However, the availability of such specifications is seriously limited at the moment: Manually writing the specifications can be prohibitively expensive and error-prone, while existing automated approaches to inferring the specifications seldom produce results that are strong enough to be used in program analysis.
In this work, we propose the DOC2SMT approach to generating strong functional constraints in SMT for library methods based on their documentations. DOC2SMT first applies NLP techniques and a set of rules to translate a method's natural language documentation into a large number of candidate constraint clauses in OCL. Then, it utilizes a manually enhanced domain model to identify OCL candidate constraint clauses that comply with the problem domain in static validation, translates well-formed OCL constraints into the SMT-LIB format, and checks whether each SMB-LIB constraint rightly abstracts the functionalities of the method under consideration via testing in dynamic validation. In the end, it reports the first functional constraint that survives both validations to the user as the result.
We have implemented the approach into a supporting tool with the same name. In experiments conducted on 451 methods from the Java Collections Framework and the Java IO library, DOC2SMT generated correct constraints for 309 methods, with the average generation time for each correct constraint being merely 2.7 minutes. We have also applied the generated constraints to facilitate symbolic-execution-based test generation with the Symbolic Java PathFinder (SPF) tool. For 24 utility methods manipulating Java container and IO objects, SPF with access to the generated constraints produced 51.2 times more test cases than SPF without the access.

Received . . .

KEY WORDS: specification generation; documentation analysis; domain model; OCL; SMT

1. INTRODUCTION

Software libraries are playing an ever more important role in constructing programs nowadays. On the one hand, code in libraries can be easily reused to accomplish common programming tasks and improve programmers' productivity. On the other hand, the libraries used in software development pose new challenges to the analysis of the resultant programs: The source code of the libraries may be hard to acquire, their compiled code may be obfuscated, and they may be written in multiple programming languages and/or implement sophisticated engineering tricks for performance reasons.

Since manually writing specifications for methods can be prohibitively expensive and error-prone, researchers have proposed various techniques in the past few years to automatically infer

*Correspondence to: State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China.

specifications for library methods, e.g., through dynamic [1, 2, 3, 4, 5] or static [6, 7] program analysis, so that their implementation details can be abstracted away and complex program analysis tasks may become possible or scalable. Another line of such work aims to infer specifications for methods from their natural language descriptions, in the form of API documentations [8, 9] or code comments [10, 11]. Pandita et al. [8] propose the ALICS approach that pioneered the application of natural language processing (NLP) techniques to specification generation for library methods. ALICS translates sentences in API documentations into logical expressions based on pre-defined shallow parsing semantic templates, and generates code-contracts from the expressions by mapping semantic classes of the predicates to programming constructs. Zhai et al. [9] present an automated model generation (AMG) technique that produces functionally equivalent model implementations for methods from API documentations. The AMG technique transforms grammatical trees of sentences to produce variants and utilizes pre-defined patterns to match tree structures and generate code snippets for the model implementations. Goffi et al. [10] propose the TORADOCU approach that extracts specifications in the form of Java conditions for exceptional behaviors from Javadoc code comments. TORADOCU applies NLP techniques to identify the subjects and related predicates of sentences describing the exception conditions, and it matches the subjects and predicates to Java code elements using approximate lexicographic matching. The Java conditions extracted with TORADOCU can be used as the oracle in testing the exceptional behaviors. Blasi et al. [11] present the JDOCTOR technique that extends TORADOCU to produce specifications for also preconditions and normal postconditions. Motivated by the observation that syntactically different terms can have a close semantics, JDOCTOR employs a neural network model to embed the semantics of words from the comments and code element identifiers, and matches the predicates to code elements with the smallest semantic distance. Among these techniques, TORADOCU infers only conditions for exceptional behaviors, while the others often produce specifications that are incomplete or hard to use in tasks like program analysis: ALICS mostly produces weak specifications such as null pointer assertions to facilitate the verification of legal usages; Model implementations produced by AMG and executable specifications inferred by JDOCTOR usually contain invocations to other methods that are still challenging to analyze.

In this paper, we propose a novel approach, named DOC2SMT, to generating *strong* functional constraints for library methods based on their documentations. Compared with specifications inferred by existing techniques mentioned above, constraints generated by DOC2SMT are expected to specify the complete normal functionalities of the methods. Particularly, the constraint generated by DOC2SMT for a method need to specify not only what results the method should produce, i.e., the postconditions, but also under what conditions the method should produce those results, i.e., the preconditions.

First, DOC2SMT applies NLP techniques to construct semantic graphs with POS tag annotations from a method's natural language documentation, and employs a set of permissive rules to translate the graphs into a large number of candidate constraint clauses in OCL. Next, DOC2SMT identifies the constraints that faithfully reflect the functionalities of a method via a novel, two-step validation process: In static validation, a manually enhanced domain model is used to help filter out syntactically ill-formed OCL expressions that do not comply with the problem domain; The well-formed OCL candidate constraints are then translated into the SMT-LIB format. During dynamic validation, whether each SMB-LIB constraint rightly abstracts the functionalities of the method under consideration is automatically checked via testing. In the end, the first functional constraint that validates successfully in both steps is reported to the user as a *valid* constraint for the method.

We have implemented the approach into a tool with the same name. To evaluate the effectiveness and efficiency of DOC2SMT, we applied the tool to generate strong functional constraints for 451 public methods defined in 24 classes from the Java Collections Framework and the Java IO library. DOC2SMT successfully produced valid constraints for 312 methods, and manual inspection confirmed that the constraints for 309 of those methods were indeed correct, i.e., both sound and complete. The average time DOC2SMT took to produce a correct constraint was merely 2.7 minutes. We have also applied the generated constraints to facilitate symbolic-execution-based test generation with the Symbolic Java PathFinder (SPF) tool. For 24 utility methods manipulating Java container

```
java.util
Class TreeMap<K,V>
java.lang.Object
  java.util.AbstractMap<K,V>
    java.util.TreeMap<K,V>
All Implemented Interfaces
..., Map<K,V>, NavigableMap<K,V>, SortedMap<K,V>, ...

A Red-Black tree based NavigableMap implementation. The map is sorted according to
the natural ordering of its keys, or by a Comparator provided at map creation
time, depending on which constructor is used.
Methods
..., containskey, get, put, replace, size, ...
```

Figure 1. Part of the documentation for class `TreeMap`.

```
1 public V replace(K k, V v)
2 Replaces the entry for the specified key only if it is currently mapped to some
      value.
3 Parameters:
4 k - key with which the specified value is associated
5 v - value to be associated with the specified key
6 Returns:
7 the previous value associated with the specified key, or null if there was no
      mapping for the key
```

Figure 2. Profile of method `replace(K k, V v)` from `TreeMap`.

and IO objects, SPF with access to the generated constraints produced 51.2 times more test cases than SPF without the access.

This paper makes the following contributions:

- We propose an approach to generating strong functional constraints for library methods from their documentations.
- We implement the approach into a prototype tool with the same name.
- We conduct experiments on methods from the Java Collections Framework and the Java IO library to evaluate the approach. Experimental results suggest the approach is able to produce high-quality constraints with reasonable efficiency and scalability.

Outline. The rest of this paper is organized as the following. Section 2 illustrates from a user's perspective how DOC2SMT generates strong functional constraints for library methods. Section 3 explains in detail the steps involved in constraint generation with DOC2SMT. Section 4 reports on the experiments we conducted to evaluate DOC2SMT. Section 5 reviews researches done in related areas. Section 6 concludes the paper.

Availability. A package with DOC2SMT's implementation and the experimental results is publicly available for download at `https://github.com/SEG-DENSE/Doc2SMT`.

2. DOC2SMT IN ACTION

In this section, we use a method to demonstrate how DOC2SMT generates strong functional constraints based on the method's documentation from a user's perspective.

A *map* is a widely-used data structure supporting fast key-based lookups. Class `java.util.TreeMap` from the Java Collections Framework implements a navigable map, i.e., a map whose elements can be easily accessed in ascending or descending key order, and it stores each pair of key

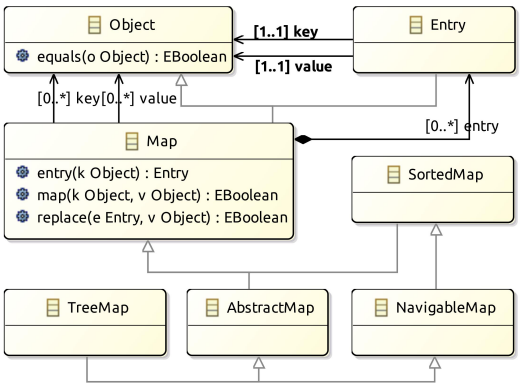

Figure 3. A domain model required for generating method `replace`'s strong functional constraint. Public methods that can be easily extracted from the class documentation are omitted for brevity reasons.

```
;declare-datatypes:
;Map (mk-map (key (Array K Bool)) (mapping (Array K V)))
;Entry (mk-entry (key K) (value V))
;declare-const:
;?p0 (Map Int Int), ?p1 Int, ?p2 Int,
;?r Int, ?_p0 (Map Int Int), t0 (Entry Int Int)
;assertions:
(= t0 ((as mk-entry) ?p1 (select (mapping ?p0) ?p1)))
(ite
 (and (select (key ?p0) (key t0)) (= (select (mapping ?p0) (key t0)) (value t0)))
 (and (= (key ?_p0) (key ?p0)) (= (mapping ?_p0) (store (mapping ?p0) (key t0) ?p2)))
 (and (= (key ?_p0) (key ?p0)) (= (mapping ?_p0) (mapping ?p0))))
```

Figure 4. The functional constraint generated by DOC2SMT for method `TreeMap::replace(K k, V v)` in SMT-LIB. A `Map` object contains a `key` set with all the keys and a `mapping` that relates each key to a value. Each `Entry` object is a key-value pair. Symbols `?p0`, `?p1`, and `?p2` refer to the three input parameters of the method, while symbols `?_p0` and `?r` refer to the receiver object and the return value at method exit, respectively.

```
From method summary:
a) if value->exists(value|value.k)
      then replace(k) else equals(self@pre) endif
b) if value->exists(value|map(k,v))
      then replace(entry(k),v) else equals(self@pre) endif
c) if value->exists(value|map(k,value))
      then replace(entry(k),v) else equals(self@pre) endif
From method return value description:
d) if not(self.contains(entry(k)))
      then result=null else map(k,result) endif
```

Figure 5. Three of the candidate OCL expressions generated by DOC2SMT from the method summary and one from the return value description for `TreeMap::replace(K k, V v)`. The conjunction of expressions c) and d) constitutes a strong functional constraint for the method.

and value as an `Entry` internally. Method `replace(K k, V v)` of the class is inherited from interface `Map` and it substitutes `k`'s currently associated value, if any, with `v`.

Developers of the class have written informative documentation for both the class and its public methods, which also serves as the specifications for the class and methods. Figure 1 shows part of the documentation for the class, and Figure 2 shows the profile information about method `replace` from the same documentation. Since the information is written in natural language, it is not readily

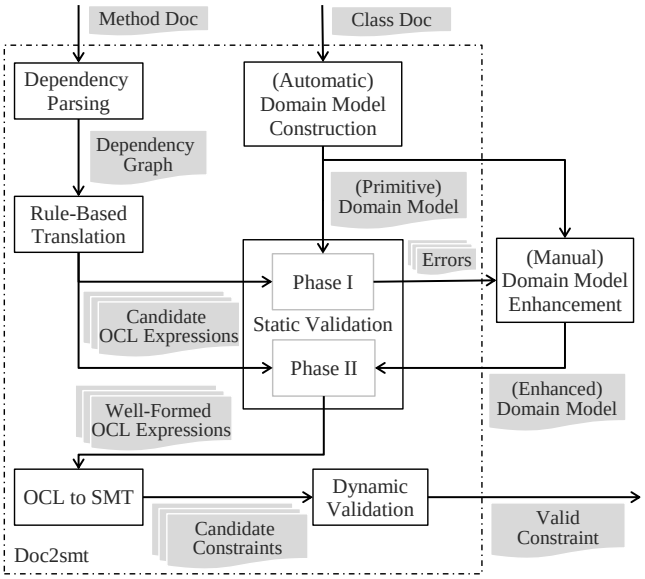

Figure 6. An overview of the DOC2SMT approach.

consumable by most other software engineering tools. In particular, such information cannot be used to support static program analysis. Taking the documentation for the class and a domain model (shown in Figure 3) as the input, DOC2SMT is able to generate in a few minutes a strong functional constraint in the SMT-LIB syntax for the method. Figure 4 gives the constraint that DOC2SMT produces for method `replace`. One thing worth noting is that, to be able to reflect the complete normal functionalities of method `replace`, the constraint generated for the method needs to differentiate two situations where the method behaves differently, i.e., when `k` is mapped to a value in the current map and when `k` is not mapped to any value in the map. More generally speaking, the generated constraint needs to encode the preconditions of the method under consideration.

During the process, DOC2SMT generates a large number of candidate constraint clauses in OCL as intermediate results. For example, Figure 5 shows some of those candidate clauses derived from the method summary and the return value description, where clauses `c)` and `d)` are correct, while clauses `a` and `b` are incorrect. The final constraint shown in Figure 4 can be derived easily from the conjunction of the two correct candidate clauses.

Three features of DOC2SMT are key to its success. First, DOC2SMT applies a set of rules to translate sentences in the input documentation into candidate constraint clauses in the OCL syntax. The rules and the translation process are permissive enough to allow the sentences to be interpreted as they were intended. Second, DOC2SMT makes use of a domain model to quickly filter out the generated candidate constraint clauses that are syntactically ill-formed. Third, DOC2SMT ensures via testing that only constraints that reflect the complete normal functionalities of methods are reported to users.

3. THE DOC2SMT APPROACH

Figure 6 shows an overview of the DOC2SMT approach. Inputs to DOC2SMT include a list of target methods and the documentations for the methods and their defining classes. For each method, DOC2SMT first employs dependency parsing to construct dependency graphs from the method's profile information (Section 3.1), and then translates the graphs into a large number of candidate constraint clauses in the OCL syntax based on a set of permissive rules (Section 3.2). Next, DOC2SMT statically checks the validity of candidate constraint clauses in two phases (Section 3.3). In phase I static validation, DOC2SMT utilizes an automatically constructed, primitive domain model to help identify domain knowledge that is necessary for characterizing the method's

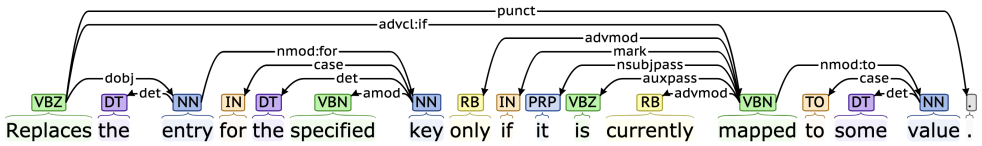

Figure 7. Dependency graph for the summary of method `replace`.

functionalities. In phase II static validation, it uses a manually enhanced domain model to prune out candidate clauses that are syntactically ill-formed. In the end, DOC2SMT translates the well-formed OCL constraints into the SMT format and checks the semantical equivalence between each remaining constraint and the method implementation via testing during dynamic validation (Section 3.5). DOC2SMT terminates upon discovering the first constraint that survives dynamic validation and outputs it as the strong functional constraint of the method.

The rest of this section details how DOC2SMT generates a strong functional constraint c for a target method m from class C, based on the documentations for C and m. Here, we use P to denote the sequence of m's input parameters and use Q to denote the sequence of m's output parameter. Q includes the return value of m, if any, and the values of the input parameters at the exit of m if they are modified by m.

3.1. Dependency Parsing of Method Summary

DOC2SMT uses the profile information of a method as the source to infer the method's functional constraint. More concretely, the profile information of a method includes the method's declaration, summary, and the descriptions of the parameters and return value, if any. For example, Figure 2 shows the profile information of method `replace` that DOC2SMT utilizes, including its declaration (line 1), its summary (line 2), as well as the descriptions of parameters (lines 4 and 5) and return value (line 7). DOC2SMT extracts the different parts of a method's profile information from the documentation of the method's containing class based on manually crafted patterns.

Next, DOC2SMT employs a dependency parser to build a dependency graph G_s from the method's summary, as was done in previous work [9, 11], and if the method returns a value, DOC2SMT also builds a dependency graph G_r based on the description of that value. The dependency graph(s) will be transformed and translated into set(s) of candidate constraint clauses in OCL afterwards. A dependency graph in natural language processing gives the grammatical structure of a sentence, where a node corresponds to a word in the sentence and may be associated with syntactic attributes like *lemma* and *part of speech* (POS) tags, while an edge reflects the typed dependency relation between words. For example, Figure 7 gives the dependency graph for method `replace`'s summary. According to the graph, word "replaces" is a verb in 3rd person singular present form (VBZ), word "entry" is the direct object (dobj) of "replaces", and word "mapped" is an adverbial clause modifier (advcl) of "replaces".

In view that the descriptions of method parameters are often noun phrases and the parameters are often referred to using those nouns in the documentation, DOC2SMT analyzes the parameter descriptions to find out the relation between parameters and their referring nouns. More concretely, given a parameter x, DOC2SMT constructs a dependency graph from x's description, identifies the root noun n of that dependency graph, and registers n as x's referring noun. Consider parameter `k` of method `replace` for example. Since the root noun in the dependency graph constructed from `k`'s description is word "key", the word is registered as the referring noun of parameter `k`. Accordingly, DOC2SMT will consider occurrences of word "key" in `replace`'s summary as possible references to `k` in its following process.

3.2. Rule-based Translation to OCL

A dependency graph gives the core text elements and their relations in a sentence. To translate a dependency graph into constraint clauses in OCL, DOC2SMT employs a rule-based technique.

Table I. Syntax of translation rules with explanations and examples. Each rule consists of a *pattern* and an *action*. A *pattern* describes a graph configuration based on nodes and edges with specific features. An *action* contains a sequence of plain texts and graph operations (`grop`). A `grop` is surrounded by a pair of square brackets (`[]`) and defines how to build a new graph using matched parts from the original graph. Note that the syntax of patterns is adopted from the Stanford CoreNLP Toolkit [12], while the syntax of actions is inspired by the work of Tuong Huan et al. [13].

Nonterminal	Syntax	Explanation
`rule`	`pattern→action`	A translation rule consists of two parts: a `pattern` that specifies the matching condition of the rule and an `action` that can be applied to produce the translations when the rule is matched.
`pattern`	`nodedec* edgedec*`	A pattern specifies matching conditions w.r.t. nodes and/or edges in a dependency graph.
`nodedec`	`{attrvalue*}=X`	Match a node, referred to as `X`, with specific attribute values. **Example:** `{tag:NN.*}=A`.
`edgedec`	`>dep?(X,Y)=E`	Match an edge, referred to as `E`, that is of type `dep` and connects two nodes `X` and `Y`. Here both `X` and `Y` can be a `nodedec`. **Example:** `>dobj(A,{tag:NN.*})=E`.
	`x,y>>dep?(X,Y)=E`	Match an edge, referred to as `E`, that is the last edge of a path connecting two nodes `X` and `Y`. The length of the path should be between `x` and `y`, and the type of `E`, if declared, should be `dep`. **Example:** `0,2>>nmod.*(A,B)=F`.
	`not edgedec`	Match a graph where no edge matches with `edgedec`. **Example:** `not >(A,{})`.
`action`	`([grop]|plaintext)*`	An `action` consists of a sequence of graph operations (`grop`s) and/or `plaintext`s. A `grop` specifies how parts identified in pattern matching are used during translation, while the `plaintext`s are directly copied into the translation results.
`grop`	`X`	Return node X declared in the corresponding pattern.
	`{attrvalue*}`	Create and return a new node with the given attributes. **Example:** `{lemma:index}`.
	`copy(X)|deepcopy(X)`	Return a copy/deep-copy of node `X`. **Example:** `copy(X)`
	$\texttt{grop}_1\texttt{-grop}_2$	Remove nodes and edges in $\texttt{grop}_2$ from $\texttt{grop}_1$ and return $\texttt{gopt}_1$.
	$\texttt{>E(grop}_1\texttt{,grop}_2\texttt{)}$	Connect the root of $\texttt{grop}_2$ to that of $\texttt{grop}_1$ via edge `E` and return $\texttt{grop}_1$. **Example:** `>E(X-Z,deepCopy(Y))`.
	$\texttt{>dep(grop}_1\texttt{,grop}_2\texttt{)}$	Connect the root of $\texttt{grop}_2$ to that of $\texttt{grop}_1$ via an edge of type `dep` and return $\texttt{grop}_1$. **Example:** `>dobj(A,B)`
	$\texttt{rep(X,grop}_1\texttt{,grop}_2\texttt{)}$	Replace node `X` in $\texttt{grop}_2$ with the result of $\texttt{grop}_1$ and return $\texttt{grop}_2$. **Example:** `rep(Z,copy(Y),X)`.

3.2.1. Translation Rules. DOC2SMT defines 26 rules to facilitate the translation from dependency graphs to OCL expressions in general cases. Table I gives the syntax of the rules, and Table II provides the complete list of general rules defined in DOC2SMT. For instance, rule TR concerns how receiver objects of method invocations are often referred to in Java documentations and how those objects should be represented in OCL expressions. Consider the summary of a method in class `TreeMap` for example. A phrase "this treemap" appearing in the summary often refers to the receiver object on which the method is invoked, and the keyword `self` should be used as its translation in OCL. Rule TR is therefore introduced to deal with such cases, where `CN`s are names of the context classes.

While the rules listed in Table II are general and useful in processing the profile information of a wide range of methods, they were not meant to be comprehensive and other rules may also

Table II. Translation rules defined in DOC2SMT. For each rule, its CATEGORY, ID, and DEFINITION.

CATEGORY	ID	DEFINITION
Condition Translation	CT1	`{}=X >({}=Y >{lemma:if}) >advcl:else{}=Z→if [Y] then [X-Y-Z] else [Z] endif`
	CT2	`{}=X >advcl:if{}=Y → if [Y] then [X-Y] else not(change()) endif`
Quantifier Introduction	QT1	`{}=X 0,2>>({lemma:all|each}=Y >nmod:of{}=Z) → [>nmod(X-Y,>det(Z,Y-Z))]`
	QT2	`{}=X 0,2>>({}=Y >det.*{lemma:all|each}=Z 0,2>>nmod.*{tag:NN.*}=M) → [M].[copy(Y)]->forAll([copy(Y)]|[X-Z-M])`
	QT3	`{}=X 0,2>>({}=Y >{lemma:some|one}=Z) → [Y-Z]->exists([copy(Y)]|[X-Z])`
Boolean Evaluation	BE1	`{tag:NN.*}=X >nsubj{tag:NN.*}=Y >{lemma:between} >conj:and{tag:NN.*}=Z → [Y].greater([copy(X)],specifyfrominclusive) and [Y].less([copy(Z)],specifytoinclusive)`
	BE2	`{lemma:less|greater}=X >nsubj{tag:NN.*}=Y >>nmod.*|dep{tag:NN.*}=Z [!>>{lemma:equal|strictly} | >>{lemma:inclusive}] → [Y].[copy(X)]([Z],specifyinclusive)`
	BE3	`{lemma:less|greater}=X >nsubj{tag:NN.*}=Y >>nmod.*|dep{tag:NN.*}=Z >>{lemma:equal}!>>{lemma:inclusive} → [Y].[copy(X)]([Z],true)`
	BE4	`{lemma:less|greater}=X >nsubj{tag:NN.*}=Y >>nmod.*|dep{tag:NN.*}=Z >>{lemma:strictly}!>>{lemma:inclusive} → [Y].[copy(X)]([Z],false)`
Passive to Active	PA	`{tag:VBN}=X {tag:NN.*}=Y {tag:VB.*}=Z >nsubj.*(X,Y) >auxpass(X,Z) → [>dobj(X-Y-Z,Y)]`
Condition Translation	CT	`{}=X 0,3>>conj:or{}=Y 0,3>>({}=Z >{lemma:if} >>{lemma:such}) → if [Z] then [Y-Z] else [X-Y-Z] endif`
Sentence Decompose	SD1	`{}=X >ccomp{}=Y → [X-Y] and [Y]`
	SD2	`{}=X 0,2>>neg{lemma:no|not}=Z:{}=X!>{{tag:VB.*|JJ.*} >{}=Z)→not([X-Z])`
	SD3	`{tag:VB.*}=X >conj:and{tag:VB.*}=Y → [X-Y] and [replace(X,Y,X-Y)]`
Noun Compo.	NC	`{tag:NN.*}=X {}=Y >amod|compound(X,Y) → [Y][X-Y]`
Noun Modifier	NM1	`{}=X 0,3>>nmod.*{tag:NN.*}=Y → [X-Y]`
	NM2	`{}=X 0,3>>nmod.*{tag:NN.*}=Y → [Y].[X-Y]`
	NM3	`{}=X 0,3>>nmod.*{tag:NN.*}=Y → [X-Y]([Y])`
	NM4	`{}=X 0,3>>nmod.*{tag:NN.*}=Y → [Y]`
Verb Dobj	VD1	`{}=X >dobj{{tag:NN.*}=Y >acl.*{}=Z) → [Y] implies [>dobj(X-Y,copy(Y))]`
	VD2	`{}=X >dobj({tag:NN.*}=Y >acl.*{}=Z) → [Y] and [>dobj(X-Y,copy(Y))]`
	VD3	`{}=X >dobj{tag:NN.*}=Y → [X-Y]([Y])`
Verb Nsubj	VN	`{}=X {tag:NN.*}=Y >nsubj(X,Y) → [Y].[X-Y]()`
Adjective Clause	AC	`{tag:NN.*}=X >acl.*{tag:VB.*}=Y → [>nmod(Y,X-Y)]`
Terminal	TE	`{}=X not >nmod.*(X,{}) → [copy(X)]`
This Reference	TR	`{lemma:CN1|CN2}=X {lemma:this|the}=Y >det(X,Y) not >(X,{word:specified}) → self`

*`specifytoinclusive`, `specifyfrominclusive`, `specifyvalue`, and `specifyinclusive` are all placeholders. They are to be replaced by parameters of the method under consideration.

be essential for DOC2SMT to support the effective translation of expressions that are specific to certain problem domains. In particular, we expect new rules need to be introduced to handle the following two types of scenarios that often occur in library documentations. First, constraints may be implied, instead of explicitly stated, in documentations and parts of a sentence may be omitted when they can be easily figured out (by human readers) from the context. In such cases, extra rules can be introduced to add the implicit or omitted information back during translation. Consider the method summary in Figure 7 for example. Verb `replace` is missing its complement "with something" and, judging from the context, "something" here should be parameter `v` of the method. To make the information complete in the translation, a new rule with a placeholder could be added. Second, certain domain specific operations can be expressed in different ways in documentations, and additional rules can help identify and equate those expressions. For example, we could define a new rule to stipulate that phrase `there is an o in c` is equivalent to phrase `c contains o` and therefore it can be translated into `c.contains(o)`. Note that DOC2SMT does not differentiate rules based on when or how they were introduced: To support the generation of functional constraints for more library methods, existing rules can be revised and/or extended and new rules can be added .

Given that translation rules like BE2 refer to unspecified values from the context via placeholders like `specifyinclusive`, DOC2SMT applies *parameter substitution* to replace such placeholders with

```
   Input: G: a dependency graph;
          Σ: a sorted set of translation rules
   Output: Π: a set of strings as the translation result
1  function TRANSLATE(G, Σ)
2      if G.nodeCount() == 1 then
3          Π ← {G.nodes().first().getLemma()} ;
4      else
5          Π ← ∅ ;
6          foreach σ ∈ Σ do
7              action ← σ.getAction() ;
8              matcher ← σ.getPattern().match(G) ;
9              while matcher.hasNextMatch() do
10                 match ← matcher.getNextMatch() ;
11                 Δ ← {action} ;
12                 foreach grop ∈ action.getGrops() do
13                     G′ ← grop.apply(G, match) ;
14                     Π′ ← TRANSLATE(G′, Σ) ;
15                     Δ ← ⋃_{δ∈Δ} ⋃_{π∈Π′} δ[grop/π] ;
16                 end
17                 Π ← Π ∪ ⋃_{δ∈Δ} δ.toString() ;
18             end
19         end
20     end
21     return Π;
```

Figure 8. Algorithm to translate a dependency graph into a list of candidate constraint clauses in OCL.

proper values after applications of the corresponding rules. For instance, `specifyinclusive` should be replaced with a boolean value that indicates whether a comparison should return true or not when the value under consideration is equal to the threshold value. Similarly, since translation rules like VD1 surround each object of a verb with a pair of parentheses, while all the parameters should be placed inside a single pair of parentheses and separated with commas if we regard the verb as denoting an operation, DOC2SMT applies *parentheses elimination* to remove redundant parentheses around parameters and adding commas between them when necessary. For example, parentheses elimination will change an invocation `map(key)(value)` to `map(key,value)`. Note that both parameter substitution and parentheses elimination are always applied at the end of the translation process, i.e., right before the OCL expressions are generated for the whole dependency graphs.

3.2.2. Translation Algorithm. When translating a piece of text, multiple rules may be applicable at the same time, and different application orders of the rules most likely will lead to distinct translation results. To avoid missing out the right translations, DOC2SMT enumerates all possible ways, instead of prematurely committing itself to a number of selected options when doing the translation. Having said that, DOC2SMT attempts first rules that tend to manipulate larger chunks of texts. For instance, rule CT3 is often applied before rule PR, since the former reorganizes several phrases while the latter replaces a single phrase.

Given a dependency graph G and a set Σ of translation rules, sorted in decreasing order of the size of texts they manipulate, function TRANSLATE shown in Figure 8 translates G into a set of OCL expressions in string by iterating through all the possible ways to apply the rules in Σ. More concretely, if G contains a single node, the function simply returns the lemma of that node (lines 2 and 3). Otherwise, the function takes each rule σ from Σ (line 6), and then repeatedly finds subgraphs in G that match the pattern specified in σ and applies the action defined in σ to produce

the translations (lines 8 through 18). When applying an $action$ to a matched subgraph, the function uses a set Δ to temporarily store the intermediate translation results produced by executing some of the graph operations defined in $action$: For each $grop$ defined in $action$ (line 12), first the $grop$ is applied on the match to produce an intermediate graph G' (line 13), then function TRANSLATE is recursively invoked to turn G' into a set Π' of strings using rules from Σ, and next each string from Π' is used to replace the corresponding $grop$ in Δ. All the translation results are collected into Π (line 17) and returned (line 21).

Applying rule-based translation to G_s produces a set E_s of candidate OCL expressions in string format. When the method returns a non-void value, another set E_r of candidates OCL expressions is generated from G_r by following the same process. Without loss of generality, we assume $E_r = \{true\}$ when the method return type is `void`.

3.3. Library Domain Model and Static Validation

Through rule-based translation, DOC2SMT often produces a large number of constraint clauses in OCL for a method. We propose to use a domain model to help determine which constraint clauses are appropriate and apply static validation in two phases to allow for an incremental preparation of the model.

Since all the containing types of the target methods and the public members of those types are clearly essential for the problem domain under consideration, DOC2SMT automatically extracts those elements from the corresponding class documentations and builds a primitive domain model based on them. More concretely, the primitive domain model contains 1) all the containing types of the target methods and their supertypes, 2) all the public members of those types, and 3) all the inheritance/implementation relations between those types. Note that DOC2SMT does not automatically follow USE-A or HAS-A relations to include more types into the primitive domain model, which helps keep the resultant model small. For example, the primitive domain model constructed for method `TreeMap.replace` will include types like `Object`, `Map`, `Entry`, and `TreeMap` and methods like `size` and `empty`, where `Entry` is a public inner class of `Map`.

While the primitive domain model contains important information about the problem domain, it may miss some of the properties and operations of those types. In particular, model elements that are not directly part of the types' public interfaces but facilitate the expression of the types' functional specifications are seldom included in the primitive domain model. For example, the documentations of quite a few methods from class `Map` mention the concept of "the entry for the specified key". While it would be desirable to have an element in the domain model to reflect this concept, it is not the case with the primitive model since no public member of the class captures the concept. To identify the useful, but missing, domain model elements, DOC2SMT statically checks the well-formedness of those OCL expressions against the primitive domain model and reports validation errors due to missing elements (i.e. properties and/or operations) from the model in descending order of their occurrence numbers in phase I static validation. Note, however, that not all reported errors are caused by problems with the primitive domain model. Due to the permissiveness of the translation rules, DOC2SMT may produce inappropriate expressions that make no sense in the problem domain. Errors reported on such expressions should simply be ignored. A user can then browse through the list of errors, decide which errors actually indicate elements to be added to the domain model, and manually enhance the model accordingly.

Once we have the manually enhanced domain model, phase II static validation is conducted to prune out OCL expressions that fail to validate. For instance, clause a) in Figure 5 will be considered invalid since it refers to a property `k` of `value`, which, however, does not exist in the domain model. Constraint clauses that do not cause any errors in phase II static validation are referred to as *well-formed* clauses.

Note that, while the well-formed clauses are in the OCL syntax, they do not necessarily adhere to the OCL standard specification, since they may contain calls to *non-pure* operations from the problem domain. For example, clause c) in Figure 5 invokes an overloaded version of method `replace`, which may modify the receiver `TreeMap` object. We use OCL expressions as the intermediate representation of the constraints to enable, with the help of an OCL expression

Table III. Syntax-directed translation from types and operations in OCL to those in SMT-LIB.

OCL	SMT-LIB
`Integer/Real/String/Boolean`	`Int/Real/String/Bool`
`SequenceType`	`(declare-datatypes (T)((List (mk-list (elements (Seq T))))))`
`SetType`	`(declare-datatypes (T)((Set (mk-set (mapping (Array T Bool))))))`
`*, +, -, /`	`*, +, -, /`
`and, or, not, implies`	`and, or, not, =>`
`src->forall(var|body)`	`(forall ((x var-type))(=> (select x in src)(body)))`
`src->select(var|body)`	`(forall ((x var-type))(and (select x in src)(ite body (select x in result)(not (select x in result)))))`
`src->exists(var|body)`	`(exists ((x var-type))(and (select x in src)(body)))`
`if c then e1 else e2 endif`	`(ite c e1 e2)`

```
1 entry(Object)
2     (= ?r ((as mk-entry) ?p1 (select (mapping ?p0) ?p1)))
```

Figure 9. SMT-LIB constraint for meta-operation `entry` from class `TreeMap`.

validator, the easy identification of valid relations among types, properties, and operations in the problem domain and to effectively prune out most invalid relations suggested by the permissive translation process OCL2SMT implements.

Let W_s and W_r be the set of well-formed clauses from E_s and E_r ($W_s \subseteq E_s, W_r \subseteq E_r$), respectively. Since a functional constraint is expected to satisfy all the requirements specified in both the method summary and the method return value description, DOC2SMT computes a set $C = \{(e_1) \texttt{ and } (e_2) \mid e_1 \in W_s, e_2 \in W_r\}$ as the set of candidate constraints for the method. Each candidate constraint $c \in C$ is a predicate on P and Q.

3.4. OCL to SMT

Candidate constraints are then translated to the SMT-LIB [14, 15] format via a syntax-directed process [16]. Table III lists the rules that DOC2SMT applies to translate the built-in OCL types and operations into declarations and expressions in SMT-LIB. Note that literal values and variables are not changed during the translation, while invocations to non-built-in operations in OCL are translated into SMT-LIB expressions that encode the semantics of the operations, using a technique similar to the one proposed by Jiang et al. [17]. Consider an operation f that has a specification $\mathcal{S}$ in SMT-LIB and is invoked in a candidate constraint for example. To translate the invocation to f to SMT-LIB, DOC2SMT first instantiates $\mathcal{S}$ using a unique variable for each input and output formal parameter of f and then binds the corresponding actual parameters with those unique variables.

The translation of operations like `entry` in Figure 5.c, however, need special treatments, since the semantics of those operations is neither readily available nor producible by DOC2SMT. We refer to such operations as meta-operations. DOC2SMT automatically identifies meta-operations and demands their semantics to be provided in SMT-LIB as part of the enhanced domain model. For example, Figure 9 gives the required semantics of meta-operation `entry` from class `Map`, where `?p0` and `?p1` are placeholders for the receiver and parameter objects, respectively, while `?r` is the placeholder for the return value of the operation.

3.5. Dynamic Validation and Valid Constraints

Given a candidate constraint c for method m, c is a *correct*, i.e., both sound and complete, functional constraint if and only if both the following two conditions C1 and C2 are fulfilled.

Condition C1 stipulates that $c(p, q)$ is a necessary condition for $m(p) = q$, i.e., $m(p) = q \Rightarrow c(p, q)$. Intuitively, condition C1 requires that, for each pair of p and q, they should satisfy constraint c if the output of $m(p)$ is q. All pairs of p and q satisfying $m(p) = q$, however, are infeasible to exhaustively enumerate in practice, therefore DOC2SMT relaxes the condition and

checks in dynamic validation whether $c(p, q)$ is necessary for $m(p) = q$ w.r.t. a limited number of samples. Specifically, DOC2SMT implements a random algorithm [18] to automatically generate a representative group T_1 of tests for m and checks, for every test $t \in T_1$, whether the input values p_1 and output values q_1 of t satisfy c, i.e., whether $c(p_1, q_1)$ holds. If that is the case, condition C1 is satisfied w.r.t. T_1.

Condition C2 stipulates that $c(p, q)$ is a sufficient condition for $m(p) = q$, i.e., $c(p, q) \Rightarrow m(p) = q$. Intuitively, condition C2 requires that, for each pair of p and q, using p as the input to invoke m should produce q as the result if p and q satisfy constraint c. Similarly, since it is often infeasible to enumerate all such pairs of ps and qs, DOC2SMT checks in dynamic validation whether $c(p, q)$ is sufficient for $m(p) = q$ w.r.t. a limited number of samples. In particular, DOC2SMT utilizes the off-the-shelf constraint solver Z3 [19] to gather a set S_1 of solutions for c and checks, for each solution $\langle p_s, q_s \rangle \in S_1$, whether the output of m upon input p_s is equal to q_s. If that is the case, condition C2 is satisfied w.r.t. S_1.

If both conditions C1 and C2 are satisfied w.r.t. the considered samples, DOC2SMT reports c as *valid*. Although dynamic validation does not provide full guarantee the valid constraints are indeed correct, because the validity is only w.r.t. a limited number of input/output pairs, most of the generated valid constraints actually turned out to be sound and complete in our experimental evaluation described in Section 4.

If given enough time, DOC2SMT may be able to report multiple valid constraints for some methods. However, the tool terminates immediately after the first valid constraint is found during dynamic validation mainly because we feel the necessity for DOC2SMT to generate multiple valid constraints is limited in practice. First, since the correctness of each valid constraint needs to be determined manually, we are more interested to have the correct constraints as the first results returned by DOC2SMT. Second, given that each valid constraint produced by DOC2SMT should hold on all the test inputs and outputs gathered via automated test generation, those valid constraints often have quite similar or even equivalent semantics, especially when a good number of tests are used for their validation. Having said that, we plan to experimentally evaluate the differences between, and the usefulness of, multiple valid constraints in the future to get a broader view of the tool's capabilities.

3.6. Implementation Details

We have implemented the approach described above into a tool, also named DOC2SMT. The tool employs the Stanford CoreNLP Toolkit [12] to build the dependency graphs and find matches for patterns in translation rules. Domain models are constructed based on the Eclipse Modelling Framework [20] (EMF). The Eclipse OCL Toolkit [21] is used to statically validate constraint clauses against domain models and translate candidate constraints into the SMT-LIB format.

It, however, is important to note that DOC2SMT is not tightly bound to any of the specific tools it uses. Other tools providing similar functionalities can be easily integrated into DOC2SMT and replace existing components.

4. EVALUATION

We conduct an experimental evaluation on DOC2SMT to address the following research questions:

RQ1: How effective is DOC2SMT? In RQ1, we carefully analyze for how many methods DOC2SMT is able to generate functional constraints and what is the quality of the generated functional constraints;

RQ2: How efficient is DOC2SMT? In RQ2, we focus on the costs of applying DOC2SMT to generate functional constraints;

RQ3: How scalable is DOC2SMT? In RQ3, we apply DOC2SMT to generate functional constraints for additional methods and measure the required amount of manual effort in terms of the numbers of translation rules and domain model elements that need to be manually added.

Table IV. Subject methods used in experiments to answer RQ4. For each CONTEXT class, the number of subject methods selected from the class (#M) and the SUBJECT CLASSES used in the class. Class `Collections` is from package `java.util`, class `IOUtils` is from package `org.apache.commons.io`, while all the other context classes are from package `org.apache.commons.collections4`.

CONTEXT	#M	SUBJECT CLASSES
`Collections`	8	`Collection`, `List`
`CollectionUtils`	2	`Collection`, `Set`, `HashSet`
`ListUtils`	5	`Collection`, `Set`, `HashSet`, `List`, `ArrayList`, `Map`, `HashMap`
`MapUtils`	1	`Map`
`IOUtils`	8	`Reader`, `Writer`, `InputStream`, `OutputStream`
TOTAL	24	-

RQ4: How useful are the generated constraints? In RQ4, we apply the functional constraints generated by DOC2SMT for all the subject methods to facilitate symbolic-execution-based test generation and assess the usefulness of the constraints in terms of how many more tests they can help produce.

4.1. Subjects

To answer RQ1 and RQ2, we choose 19 common container classes from the Java Collections Framework[†] (JCF) as our subject classes. The container classes are notorious for their complexity that affects program analysis. To answer RQ3, we choose 5 frequently used classes from the `java.io` package of JDK as additional subject classes. These IO classes are widely used in applications to access external data. All the 24 subject classes are among the most widely used libraries and they significantly increase the complexity of program analysis tasks. Columns CLASS and #M of Tables V and VII list the 24 subject classes and the number of subject methods chosen from each class. To answer RQ4, we gather in total 24 utility methods from the JCF and the Apache Commons project[‡] that manipulate only objects of the subject classes studied in RQ1, RQ2, and RQ3. Table IV gives basic information about the context classes of those methods and the subject classes used in each context class.

4.2. Experimental Protocol

To answer RQ1 and RQ2, for each subject method from the 19 Java collections classes, we first use a simple script to extract its profile information from the corresponding class documentation and introduce necessary rules to DOC2SMT so that it can effectively translate the profile information of the subject methods into OCL. Then, we construct a primitive domain model by automatically extracting types and their public members from the class documentations, identify important missing elements from the primitive domain model by running phase-I static validation and gathering missing element errors reported by DOC2SMT whose occurrence numbers were greater than 10. Next, we add model elements that are truly missing to produce an enhanced domain model and obtain syntactically well-formed OCL candidate constraint clauses by running phase-II static validation. In the end, we apply DOC2SMT to translate the obtained OCL constraints to SMT-LIB and dynamically validate them against the corresponding method implementations via testing.

To answer RQ3, we repeat the above process on methods from the 5 Java IO classes, but using all the available translation rules and the enhanced domain model from the previous experiments as the basis. Such setting is in line with how DOC2SMT is supposed to be used in practice. That is, both the translation rules and the domain model used in earlier applications of DOC2SMT could be accumulated and reused to make future applications of the tool less expensive.

We decide whether a valid SMT-LIB constraint produced by DOC2SMT is *correct* or not through manual inspection. We are aware that manual assessment may cause a major threat to the construct

[†] `https://docs.oracle.com/javase/8/docs/technotes/guides/collections/`
[‡] `https://commons.apache.org/`

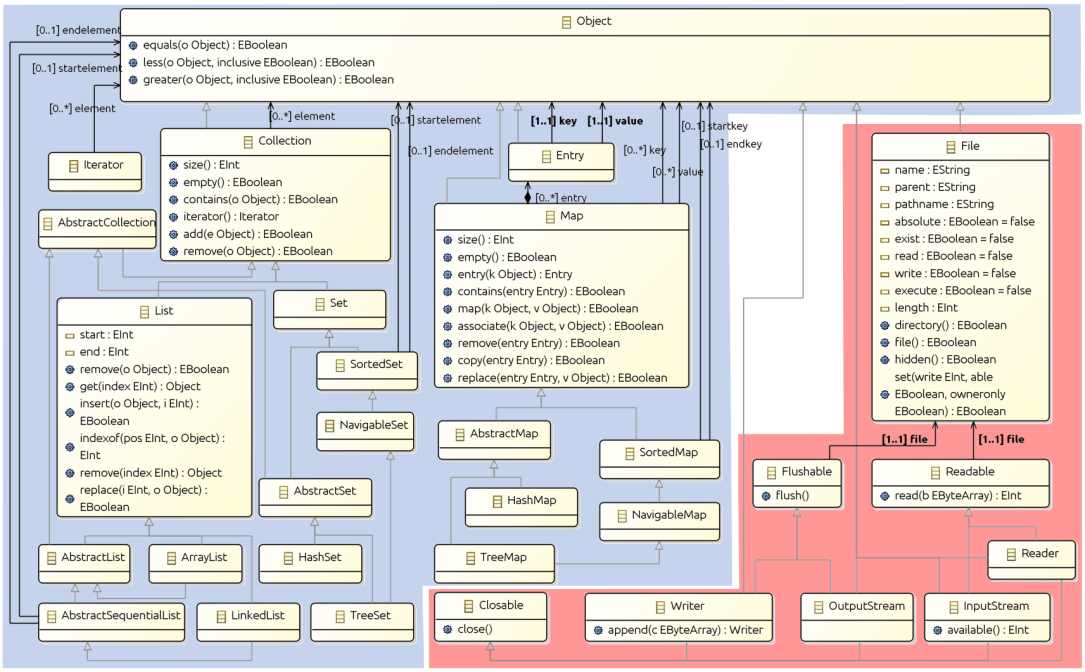

Figure 10. Enhanced domain model used in the experiments. Note that public methods defined in the types have been omitted for space reasons.

validity of our findings. To mitigate the threat, two authors independently examine the quality of each valid constraint reported by DOC2SMT, and a constraint is only marked as *correct* if both authors agree that the constraint properly captures all the normal functionalities of the corresponding method.

In each application of DOC2SMT in our experiments, the tool spends at most 2 minutes on each phase of static validation, it generates at most 100 different tests for each method, it finds at most 100 solutions for each candidate constraint in dynamic validation, and the time out for each invocation to Z3 is set to 20 seconds. We record the following measures for the run of DOC2SMT on each method:

#OCL: number of OCL constraint clauses generated from method profile information;
$\#\text{OCL}_{\text{WF}}$: number of well-formed constraint clauses in OCL retained after static validation;
$\#\text{OCL}_{\text{DV}}$: number of well-formed candidate constraints that have been dynamically validated until a valid one is found;
#SOLU: number of solutions the Z3 solver produces for all the candidate constraints;
#TEST: number of tests generated for all the methods;
T: wall-clock running time of DOC2SMT;
T_{G}: wall-clock time for OCL expression generation;
T_{SV}: wall-clock time for static validation;
T_{DV}: wall-clock time for dynamic validation;

To answer RQ4, we employ two symbolic-execution-based test case generation tools, namely Symbolic Pathfinder (SPF) and Enhanced SPF (ESPF), to generate test cases for the 24 utility methods. SPF [22] is built on the top of the Java PathFinder (JPF) model checker, while ESPF [23] extends SPF and is able to make use of available method specifications in SMT during its analysis. In particular, ESPF modified the interpretation of `invoke` instructions in SPF in such a way that, when an invocation to a method with specification in SMT is encountered, the specification is incorporated directly in the same way as is done during the translation from OCL to SMT (Section 3.4).

Each test case generation session with (E)SPF starts with a single initial test for the method under consideration in our experiments. During test generation, (E)SPF gathers the path conditions of the executed tests, flips the values of their branch conditions to form new path conditions, and sends

Table V. Results produced by DOC2SMT on 19 Java collections classes. For each subject CLASS, the number of methods processed (#M), the numbers of valid (#V) and correct (#C) functional constraints in SMT-LIB produced (SMT), and the other measures defined in Section 4.2. All times are in seconds.

CLASS	#M	SMT		#OCL	S-V		D-V			TIME			
		#V	#C		$\#OCL_{WF}$	#M'	$\#OCL_{DV}$	#SOLU	#TEST	T	T_G	T_{SV}	T_{DV}
`Collection`	19	9	9	604K	160	13	32	644	900	2253.6	0.7	3.5	2246.7
`AbstractCollection`	14	8	8	569K	125	12	31	602	1200	3073.1	15.1	24.4	3033.6
`Set`	16	11	11	2737K	142	13	25	812	1100	2802.0	0.7	113.0	2576.0
`AbstractSet`	3	2	2	33K	19	2	2	133	200	444.2	0.3	2.6	441.3
`HashSet`	9	8	8	65K	37	8	8	530	800	1354.2	0.3	6.0	1342.3
`SortedSet`	7	5	5	862K	16	5	5	410	500	1085.3	0.5	39.4	1045.5
`NavigableSet`	15	14	13	4657K	137	14	22	1193	1400	3047.6	11.9	390.2	2645.5
`TreeSet`	27	24	23	4831K	192	25	34	1884	2400	5275.5	1.7	380.0	4515.4
`List`	28	17	17	13901K	206	21	28	704	1418	1720.4	1.6	377.0	966.4
`AbstractList`	16	10	10	1552K	82	12	18	376	824	864.2	2.0	58.3	803.9
`AbstractSequentialList`	7	5	5	98K	33	6	12	191	331	526.9	0.5	4.5	522.0
`ArrayList`	31	16	16	14415K	234	20	27	758	1362	1799.8	1.2	377.0	1045.9
`LinkedList`	40	34	33	13864K	483	37	48	1473	3152	2538.5	1.7	297.8	1942.9
`Map`	25	15	15	10077K	222	18	21	908	1439	3263.1	10.1	425.8	2411.5
`AbstractMap`	16	12	12	797K	93	13	13	686	1158	2084.6	1.6	79.9	2003.2
`HashMap`	24	15	15	10141K	213	18	23	839	1445	4019.8	9.2	410.1	3199.6
`SortedMap`	9	7	7	840K	17	8	8	369	700	944.5	0.9	32.6	911.0
`NavigableMap`	21	19	19	5204K	159	19	19	1478	1900	4188.8	2.8	780.7	3405.3
`TreeMap`	40	33	33	8506K	288	35	39	2223	3245	7325.5	3.7	948.3	5428.9
Total	367	264	261	93753K	2858	299	415	16213	25474	48611.6	66.5	4751.1	40486.9

each new path condition to the Z3 solver. If the solver can find a solution to the new path condition and a test can actually be constructed using the solution to exercise a new path, a new test has been generated. In this way, (E)SPF is able to produce a group of new tests, each covering a distinct path than the initial test does. Here, we configure SPF and ESPF to run on each method until either 100 different tests have been generated or a 2-minute time limit has been reached, and we record the number of new tests generated. Our choice of such stop criterion is motivated by the "small-scope hypothesis" [24], which claims that many defects can be triggered with small inputs and witnessed using short executions.

Our experiments were conducted on a desktop computer running Ubuntu 16.04 on a Intel Core i7-6700 CPU (3.4GHz) and 16G RAM.

4.3. Experimental Results

In this section, we report the experimental results and answer the four research questions.

4.3.1. RQ1: Effectiveness. Table V shows that, in total, DOC2SMT was able to generate valid constraints for 264 of the 367 subject methods, and 261 of the 264 valid constraints were unanimously considered correct. In other words, functional constraints generated by DOC2SMT achieved an overall recall of 71.1% and an overall precision of 98.9%. Such results strongly suggest that DOC2SMT is not only highly applicable but also effective in producing high quality constraints for library methods.

The constraints DOC2SMT generated for methods `NavigableSet::descendingSet`, `TreeSet::descendingSet` and `LinkedList::pollLast` were valid but incorrect. The first two methods should return a set of the same elements from the original set but with a reversed navigation order. Since we did not provide the meta-operation to properly test the equality of navigable set objects, DOC2SMT had to make do with the semantics of a more general comparison operator defined for set objects during dynamic validation. The criteria for valid constraints were therefore weaker than what they should be, allowing the incorrect constraints to survive the dynamic validation. The third method should retrieve and remove the last element of a list, or return `null` if the list is empty. For the method, DOC2SMT failed to generate any test that can expose the discrepancy between the result constraint and the method's semantics in its current settings. While DOC2SMT also generated the

correct constraint for the method, the constraint was further down the list of all candidates and therefore missed by the tool.

DOC2SMT failed to produce any valid constraint for 103 methods. We manually checked those methods and identified four reasons for the failures: 1) DOC2SMT failed to produce any well-formed constraint clause for 67 methods due to *incomplete domain model*. More concretely, iterator types like `Spliterator` and `Listiterator`, function types like `BiFunction` and `Predicate`, and other types with specifications that are highly implementation-dependent like `Comparator` were not included in the enhanced domain model, which caused unsuccessful generation of functional constraints with DOC2SMT on 19, 21, and 27 methods, respectively. For example, since type `Comparator` is not part of the domain model, while method `TreeMap::comparator` returns an object of type `Comparator`, DOC2SMT could not generate any candidate constraint clause in OCL for the method that passes the static validation; 2) For 10 other methods, while DOC2SMT was able to generate the correct candidate constraints, the constraints were pruned out during dynamic validation since the Z3 solver was not able to find any solution to them. For example, while the generated constraint for method `TreeMap::values` correctly stipulates that the result collection contains all the symbols in the map that are associated with a particular key, it is regarded *invalid* since Z3 returned `unknown` when solving the constraint; 3) Due to *limitations in the linguistic analysis*, DOC2SMT was not able to properly handle the method profile information and produced only invalid constraints on 11 methods. For example, when dealing with method `TreeMap::replace(K key, V oldvalue , V newvalue)`, DOC2SMT did not correctly understand the corresponding documentation and failed to generate the correct constraint "if map(k,oldvalue) then replace(entry(k),newvalue) else equals(self@pre) endif"; One possible way to (at least partially) overcome this limitation is to utilize the semantic information derived from a method's implementation to help infer the method's semantics, as was proposed by Blasi et al. [11]. 4) DOC2SMT failed to generate any constraint that encode the complete semantics on 15 methods, because some aspects of the corresponding semantics was only implied, rather than explicitly provided, in the documentations. As the result, even if DOC2SMT was able to generate constraints that faithfully reflect the documentations, the constraints were invalidated during dynamic validation. For example, the documentation for method `Set::toarray` explicitly requires that the result should contain all elements in the set, but not that the result should only contain elements from the set. Therefore, the constraints generated by DOC2SMT reflect only the weak specification given in the method documentation and failed to validate dynamically because they admit solutions that do not comply with the actual dynamic behaviors of the method.

As listed in Table V, DOC2SMT generated over 90 million candidate constraint clauses for the 367 subject methods, among which only 2858 constraint clauses for 299 methods validated successfully against the enhanced domain model and were well-formed. After dynamically checking 679 (=415+264) constraints, DOC2SMT reported valid results for 264 methods. These numbers suggest that DOC2SMT explored a fairly large space in constructing candidate constraint clauses, and its remaining steps were effective in pruning out the invalid constraint clauses and reporting only the ones of high quality.

DOC2SMT *generated valid constraints for 264 of the 367 subject methods and 261 of those generated constraints were correct, achieving a recall of 71.1% and a precision of 98.9% overall.*

4.3.2. RQ2: Efficiency. Since functional constraint generation with DOC2SMT is not fully automated, we examine the efficiency of the tool from two different aspects, i.e., the time costs for running the DOC2SMT tool and the manual effort required to prepare some of the necessary inputs.

Table V also shows the amount of time DOC2SMT took to produce the valid constraints and its breakdown into the amount of time spent on each of the three main steps, i.e., candidate constraint generation, static validation, and dynamic validation. It took DOC2SMT around 13.5 hours in total to produce the results, averaging to 42.6 minutes for each class or 3.1 minutes for each correct constraint. Among the three main steps, dynamic validation is by far the most time-consuming,

Table VI. Additional translation rules for translating the documentations of the 19 Java collection classes.

CATEGORY	ID	DEFINITION
Pronoun Replacement	PR	{}=X {tag:NN.*}=Y {lemma:it|they}=Z 0,2>>(X,Y) 0,2>>(X,Z) not 0,2>>(Y,Z) → [rep(Z,deepCopy(Y),X)]
Quantifier Introduction	QT	{}=X 0,2>>({}=Y >det.*{lemma:all|each}=Z ! 0,2>>nmod.*{}) → [copy(Y)]->forAll([copy(Y)]|[X-Z])
Sentence Decompose	SD	{tag:VB.*}=X >conj:but{{tag:VB.*}=Y >neg{}=Z) → [replace(Y,X-Y,Y-Z)]
Boolean Evaluation	BE1	{tag:NN.*}=X >nsubj{lemma:index}=Y >{lemma:between} >conj:and{tag:NN.*}=Z → [copy(X)] <= [Y] and [Y] < [copy(Z)]
	BE2	{lemma:range}=X >nsubj{tag:NN.*}=Y >nmod:from({tag:NN.*}=Z >nmod:to{tag:NN.*}=M) → [Y].greater([copy(Z)],specifyfrominclusive) and [Y].less([copy(M)],specifytoinclusive)
	BE3	{tag:NN.*}=X >amod.*{lemma:greatest|least}=Y 0,2>>amod.*{}=Z : {}=Y ! == {}=Z → [X-Y-Z]->select([copy(X)]|[>nsubj(Z,X-Y-Z)])->collect(x|x.oclAsType(Integer))->[copy(Y)]().oclAsType(Object)
Adjective Clause	AC	{}=X >ref{lemma:whose} >acl.*{{tag:NN.*}=Z >nsubj{}=M) → [>nsubj(Z-M,>nmod:of(M,copy(X)))]
Special Structure	SS1	{lemma:be}=X {lemma:there}=Y {tag:NN.*}=Z >expl(X,Y) >nsubj(X,Z) → contains([Z])
	SS2	{tag:VB.*}=X >=M {lemma:element}=Y >nmod:at {}=Z → [>M(X-Y-Z,>dobj({lemma:get;tag:NN},Z))]
Implicit Constraint	IC1	{lemma:replace}=X >dobj {lemma:entry}=Y ! >nmod.* {} → [X-Y]([Y],specifyvalue)
	IC2	{lemma:index}=X >nmod:of {}=Y → [>dobj({lemma:indexof;tag:NN},Y)]
	IC3	{lemma:element}=X >nmod:at {}=Y → [>dobj({lemma:get;tag:VB},Y)]
	IC4	{word:contained}=X >nsubj {}=Y >nmod.* {tag:NN.*}=Z → [>nmod(Y,Z)]
	IC5	{lemma:view}=X 0,2>>nmod:of {lemma:collection|.*list|.*set}=Y → result.element->forAll(element|contains(element)) and [Y]
	IC6	{lemma:view}=X 0,2>>nmod:of {lemma:.*map}=Y → result.entry->forAll(entry|contains(entry)) and [Y]
	IC7	{lemma:view}=X 0,2>>nmod:of {tag:NN.*}=Y > {lemma:collection|set} → result.element = [copy(Y)]
	IC8	{lemma:set}=X >ref {lemma:whose} >acl:relcl ({}=Y >nsubj {}=Z) → [X-Y].[copy(Z)]->forAll([copy(Z)]|if [Y] then result.contains([copy(Z)]) else not(result.contains([copy(Z)])) endif)
	IC9	{lemma:map}=X >ref {lemma:whose} >acl:relcl ({}=Y >nsubj {}=Z) → [X-Y].[copy(Z)]->forAll([copy(Z)]|if [Y] then result.containskey([copy(Z)]) else not(result.containskey([copy(Z)])) endif)

accounting for 83.3% of the overall generation time, since it involves generating and running many tests on each method.

Manual effort was required in our experiments to devise the domain specific translation rules, to enhance the primitive domain model, and to craft SMT constraints encoding the semantics of meta-operations. To support the translation of the methods' profile information into OCL, besides of using 23 existing rules from Table II, we had to introduce 18 extra translation rules, as listed in Table VI. In other words, we had to manually prepare 43.9% of the translation rules required in these experiments. DOC2SMT constructed a primitive domain model with 22 types and 367 public operations automatically extracted from the documentations and phase-I static validation of OCL constraint clauses against the primitive domain model reported 261 missing element errors, of which 44 had occurrence numbers greater than 10. Since manual examination revealed that 37 of the errors indeed reflected elements that should be part of the domain model but were missing, we added 24 properties and 13 operations to the model and produced an enhanced domain model for the classes, as shown in the blue part of Figure 10. We also identified 26 meta-operations in the enhanced domain model and had to manually specify their semantics in SMT-LIB.

In the end, 197, 244, and 84 methods where DOC2SMT produced correct constraints required at least one extra translation rule, one added domain model element, and the semantics of one new meta-operation for DOC2SMT to produce their results, respectively. In view that the generated constraints were of high quality and that the results of the manual effort can be reused in processing other Java library documentations in the future, we consider the overall costs for functional

Table VII. Results produced by DOC2SMT on 5 IO classes.

CLASS	#M	SMT		#OCL	S-V		D-V			TIME			
		#V	#C		$\#OCL_{WF}$	#M'	$\#OCL_{DV}$	#SOLU	#TEST	T	T_G	T_{SV}	T_{DV}
`File`	50	21	21	928K	32	24	26	2100	2100	319.0	55.5	72.7	190.7
`Reader`	10	6	6	2K	14	6	6	600	600	79.5	18.8	2.0	58.7
`Writer`	10	10	10	2K	19	10	10	1000	1000	116.6	2.0	1.0	113.7
`InputStream`	9	6	6	8371K	17	6	6	600	600	260.3	3.5	185.2	71.5
`OutputStream`	5	5	5	314K	9	5	5	500	500	76.5	1.1	13.1	62.3
Total	84	48	48	9617K	91	51	53	4800	4800	851.9	80.9	274	496.9

Table VIII. Additional translation rules for translating the documentations of the 5 Java IO classes.

CATEGORY	ID	DEFINITION
Sentence Decompose	SD1	`{tag:VB.*}=X >conj:and{tag:VB.*}=Y` → `[Y]`
	SD2	`{tag:VB.*}=X >conj:and{{}=Y >cop{tag:VB.*})` → `[X-Y] and [replace(X,Y,X-Y)]`
	SD3	`{}=X >cop{}=Y >nsubj{}=Z` → `[Z].[Y][X-Y-Z]`
	SD4	`{tag:VBN}=X >nsubjpass{}=Y >auxpass{tag:VB.*}=Z` → `[Y].[X-Y-Z]`
	SD5	`{tag:VBN}=X >nsubjpass{}=Y >advmod{tag:JJ}=Z` → `[Y].[Z][X-Y-Z]`
Noun Mod.	NM	`{}=X 0,3>>nmod.*{tag:NN.*}=Y` → `[Y].[X-Y]()`
Special Structure	SS1	`{}=X >dobj {tag:NN.*}=Y` → `[Y].[X-Y]()`
	SS2	`{tag:NN.*}=X >amod {}=Y >amod {}=Z` → `[Z]`
	SS3	`{} >nsubj({lemma:number}=X >det|amod {}=Y >nmod:of {tag:NN.*})` → `[Y]`
Implicit Constraint	IC1	`{tag:VBN}=X >aux {lemma:can}=Y >auxpass {tag:VB.*}=Z` → `[>xcomp(X-Y,{lemma:able;tag:JJ})]`
	IC2	`{tag:VB.*}=X >xcomp {lemma:able;tag:JJ}=Y` → `[X-Y][Y]`
	IC3	`{}=X >ccomp ({}=Y >mark {tag:IN}=Z)` → `[Y]`

constraint generation with DOC2SMT as moderate. We leave a more systematic and quantitative analysis of DOC2SMT's efficiency for future work.

DOC2SMT *generated correct constraints for 261 methods in 13.5 hours, averaging to 3.1 minutes per correct constraint. A moderate amount of manual work was required to produce such results.*

4.3.3. RQ3: Scalability. Table VII gives the results produced by DOC2SMT on 5 Java IO classes in the same measures as reported in Table V. In total, DOC2SMT was able to generate valid constraints for 48 of the 84 subject methods, and all of those valid constraints were unanimously considered correct, producing an overall recall of 57.1% and an overall precision of 100.0%. It took DOC2SMT 14.2 minutes in total to produce the results, averaging to 2.8 minutes for each class or 17.7seconds for each correct constraint. During the process, DOC2SMT generated over 9 million candidate constraint clauses for the 84 subject methods, among which only 91 constraint clauses for 51 methods validated successfully against the enhanced domain model and were well-formed. After dynamically checking 102 (=53+48) constraints, DOC2SMT reported valid results for 48 methods. The average generation time per valid constraint on IO classes was significantly shorter than that on collections classes. We conjecture such difference is due to two main reasons. First, most of the generation time with DOC2SMT is spent on dynamic validation, while DOC2SMT generated many more well-formed constraint clauses in OCL for methods from collections classes and needed more time to prune out the invalid ones via testing. Second, constraints generated for methods from collections classes were more complex than those for methods from IO classes, and therefore using Z3 to solve those constraints took longer time.

The effective translation of the methods' profile information into OCL utilized in total 31 translation rules, among which 19 were existing (i.e., from Tables II or VI) and 12, or 38.7%, were new. Table VIII lists all the newly introduced rules. Recall that we had to manually prepare 43.9% of the translation rules required in experiments on methods from collections classes. The different percentages clearly show that many of the translation rules can be reused in processing new documentations, and that the required effort for manually preparing the extra translation rules decreases significantly in later applications of DOC2SMT.

Table IX. Comparison between SPF and ESPF in test case generation.

METHOD	%COV	ESPF			SPF		
		#TEST	$\Delta_{\%COV}$	T (MS)	#TEST	$\Delta_{\%COV}$	T (MS)
`Collections.indexedBinarySearch`	33.3%	15	66.7%	659	6	8.4%	120
`Collections.reverse`	25.0%	2	60.7%	175	0	0	71
`Collections.max`	57.1%	9	32.2%	4741	0	0	68
`Collections.rotate`	12.5%	12	50.0%	33352	9	31.3%	634
`Collections.rotate`	27.3%	13	45.1%	25084	9	45.1%	233
`Collections.indexOfSubList`	43.2%	98	56.8%	4442	0	0	42
`Collections.lastIndexOfSubList`	33.3%	93	66.7%	9094	0	0	41
`Collections.disjoint`	47.2%	19	41.1%	3070	0	0	42
`CollectionUtils.containsAll`	12.5%	96	87.5%	120084	0	0	72
`CollectionUtils.containsAny`	36.2%	17	38.4%	2388	0	0	81
`ListUtils.intersection`	11.3%	7	19.0%	98181	0	0	89
`ListUtils.subtract`	38.6%	99	61.4%	10347	0	0	52
`ListUtils.union`	45.8%	0	0	845	0	0	54
`ListUtils.retainAll`	32.5%	99	67.5%	45248	0	0	40
`ListUtils.removeAll`	47.6%	99	52.4%	24428	0	0	52
`MapUtils.invertMap`	32.5%	5	35.6%	3782	0	0	49
`IOUtils.toByteArray`	27.3%	12	61.4%	1055	0	0	47
`IOUtils.writeChunked`	10.8%	96	89.2%	8723	0	0	53
`IOUtils.copyLarge`	55.6%	83	35.3%	10624	0	0	56
`IOUtils.copyLarge`	40.2%	7	38.0%	24357	0	0	88
`IOUtils.contentEquals`	12.5%	94	87.5%	7945	0	0	55
`IOUtils.skip`	35.2%	85	36.9%	11543	0	0	69
`IOUtils.read`	66.7%	89	27.8%	9851	0	0	51
`IOUtils.readLines`	22.1%	79	54.2%	16885	0	0	64
Total	33.6%	1228	50.5%	476903	24	3.6%	2223

DOC2SMT constructed a primitive domain model with 9 types and 84 public operations automatically extracted from the documentations. Phase-I static validation of OCL constraint clauses against the primitive domain model reported 184 missing element errors, of which 25 had occurrence numbers greater than 10. Since manual examination revealed that 24 of the errors indeed reflected elements that should be part of the domain model but were missing, we added 10 properties and 14 operations to the model and produced an enhanced domain model for the classes. Figure 10 shows the whole enhanced domain model that we produced at the end of the experiments, where the elements we added to support the handling of the 5 Java IO classes are marked in red. We also identified and provided specifications in SMT-LIB for 15 meta-operations. The numbers of manually added domain model elements and manually specified meta-operations in experiments on IO classes are 64.9% and 57.7% of those numbers in experiments on collections classes. Such result complies well with the fact that collections types are more complex than IO types. In particular, the average number of methods defined in each IO type is 12.0 (=84/7), which is 68.7% of the average number of methods defined in each collections type, i.e., 17.5 (=367/21).

Manual effort was necessary for DOC2SMT to generate all of the 48 correct constraints. Specifically, 37, 48, and 23 of the 48 methods required at least one extra translation rule, one added domain model element, and the semantics of one new meta-operation for DOC2SMT to produce their results, respectively.

The percentage of translation rules that need manual preparation decreases significantly in later applications of DOC2SMT*; The amount of other manual effort required when applying* DOC2SMT *is in proportion to the complexity of the involved classes.*

4.3.4. RQ4: Usefulness in test generation. Table IX lists, for each utility method, the code coverage achieved by the single input test (%COV) as well as the number of new tests generated (#T), the *extra* percentage of code covered by the new tests ($\Delta_{\%COV}$), and the total generation time in milliseconds (T), using SPF and ESPF, respectively. Here, code coverage is measured at the level of bytecode instructions.

Overall, ESPF generated 1228 new tests for the 24 utility methods, increasing the overall code coverage from 33.6% to 84.1%, while SPF was only able to generate 24 new tests for 3 methods,

increasing the overall code coverage from 33.6% to 37.2%. Particularly, SPF was not able to generate any new test on 21 methods, and it could not generate enough new tests to cover more than 80% of the method code on any of the remaining 3 methods. In contrast, ESPF generated new tests to cover over 80% of the code for 15 methods, and it even achieved 100% code coverage with the new tests on 9 methods. The reason for such huge difference is that SPF always uses concrete, instead of symbolic, values when encountering objects for which it cannot generate symbolic expressions. Under such a circumstance, no symbolic path condition will be constructed from an execution of the code, therefore no new inputs can be generated to drive the execution along a different path. Functional constraints produced by DOC2SMT, however, enabled ESPF to symbolically interpret method invocations on container and IO objects and to gather symbolic path conditions corresponding to various executions, which in the end enabled the generation of many more tests for the utility methods. The test generation time with ESPF was 214.5 times of that with SPF, which is understandable since little symbolic execution, and analysis in consequence, was done during the execution of the latter.

Such experimental results provide initial, but clear, evidence that the constraints generated by DOC2SMT can effectively facilitate symbolic-execution-based test case generation.

Functional constraints generated by DOC2SMT *helped symbolic-execution-based test generation produce 51.2 times more new tests for 24 utility methods.*

4.4. Limitations

We observe several important limitations in generating strong functional constraints for library methods with DOC2SMT.

The first limitation has to do with the types of information DOC2SMT utilizes in constraint generation. We focus on deriving strong functional constraints from method profiles only in this work, while other parts of the methods' documentations often contain valuable information about the semantics of those methods too. Therefore, one interesting direction we plan to explore in the future is to extend the DOC2SMT approach to take the whole documentations into account in generating the constraints. The second limitation is due to inconsistent or even incorrect documentations. DOC2SMT assumes natural language documentations provide correct specifications for library methods. However, previous research has shown that a significant amount of documentations are inconsistent or incorrect w.r.t. the source code [25, 26, 27]. To overcome this limitation, existing techniques for detecting and repairing such inconsistent or incorrect documentations could be integrated with DOC2SMT so that constraint generation is only applied to documentations that faithfully reflect the semantics of their corresponding methods. The third limitation is related to the expressiveness of OCL and SMT-LIB as modeling languages. DOC2SMT first translates natural language documentations to OCL and then produces the final constraints in the SMT-LIB format, so it is only natural that DOC2SMT cannot generate constraints beyond the expressing power of the two modeling languages. Note, however, that, since DOC2SMT is designed and implemented in a way to enable easy replacement of existing modules with new ones, we do not expect it to be difficult to incorporate more powerful modeling languages and their analysis tools, when they become available, into DOC2SMT.

5. RELATED WORK

We review in this section research studies that have been done in the area of specification inference and are closely related to this work.

Various techniques have been proposed to infer specifications for programs. Some approaches infer API specifications by statically mining API usage patterns from the client code and constructing common constraints as the specifications [28, 6, 29], while others detect invariants by dynamically running a program and using machine learning algorithms to analyze the execution traces [1, 2]. Recently, researchers proposed new techniques to infer specifications in a "guess and

validate" fashion [30, 31, 32]. Such a technique executes a target program with initial inputs, guesses invariants from the execution traces, looks for counter-examples to invalidate the guessed invariants, and adapts the guesses based on the counter-examples. The process is repeated until the guessed invariants would validate, i.e., no counter-examples can be found. There have also been approaches that try to improve specification quality by combining dynamic analysis with static analysis [3] or by exploiting other information, e.g., programmer-written contracts and second-order constraints, in programs [4, 33].

Much work has been done to generate specifications from code comments and documentations written in natural language. ALICS [8] is the first approach that analyses API documents to generate code contracts. It translates sentences in API documentations into logical expressions based on pre-defined shallow parsing semantic templates, and generates code-contracts from the expressions by mapping semantic classes of the predicates to programming constructs. Tan et al. [34] propose the @tComment approach to determine whether the exceptions thrown by a target API when some parameters are `null` are consistent with the document description. Goffi et al. [10] propose the TORADOCU approach that extracts specifications in the form of Java conditions for exceptional behaviors from Javadoc code comments. TORADOCU applies NLP techniques to identify the subjects and related predicates of sentences describing the exception conditions, and it matches the subjects and predicates to Java code elements using approximate lexicographic matching. The Java conditions extracted with TORADOCU can be used as the oracle in testing the exceptional behaviors. DASE [35] extracts constrains that should be satisfied by valid arguments to help symbolic execution engines explore different execution paths. Zhai et al. [9] propose to construct model implementations for Java APIs based on documentations. The model implementations are simpler compared to the original ones and hence easier to analyze. Blasi et al. [11] present the JDOCTOR technique that extends TORADOCU to produce specifications for also preconditions and normal postconditions. Motivated by the observation that syntactically different terms can have a close semantics, JDOCTOR employs a neural network model to embed the semantics of words from the comments and code element identifiers, and matches the predicates to code elements with the smallest semantic distance.

While these techniques extract useful results, the specifications they infer are often weak, in the sense that they are sound but incomplete, or hard to use in tasks like static program analysis. To the best of our knowledge, DOC2SMT is the first NLP-based approach to generating specifications that are good enough to be directly utilized by static program analysis techniques like symbolic execution, which hold a high expectation for the soundness and completeness of its input specifications. Compared with method implementations, method documentations usually contain high level description of methods' functionalities. Correspondingly, compared with inferring specifications by statically or dynamically analyzing the method source code, generating specifications from code documentations has the advantage of not having to deal with the low level details.

DOC2SMT was inspired by the work that uses model-driven techniques to facilitate the documentation analysis. Text2Test [36] builds models from use case specifications and it facilitates the revision of use cases based on the construction and analysis of models. The UMTG [37] approach applies NLP techniques to generate use case models from specifications and derives system test cases from the generated models. OCLgen [38] complements UMTG by automatically generating the OCL constraints to capture the pre- and post-conditions of use case steps. GUEST [13] is a rule-based approach to extract goal and use case models from natural language requirements documents. Bajwa et al. [39, 40] extract OCL constraints from documents based on Semantic Business Vocabulary and Rules (SBVR).

Compared with these techniques, DOC2SMT extracts constraint clauses in OCL from method documentations that programmers often write in plain English, prunes out syntactically ill-formed constraint clauses with the help of a manually enhanced domain model, and translates the well-formed OCL constraint clauses to the SMT-LIB format. Our idea of using a domain model to manage domain knowledge in this paper was inspired by UMTG, and the design of DOC2SMT's translation rules was inspired by the goal extraction rules proposed in GUEST.

Given the wide use of UML/OCL as modeling language in industry, various studies have investigated the translation of UML/OCL models to other specification languages for verification or analysis purposes. For example, Soeken et al. [41] turn the verification of UML/OCL models into a SAT problem by encoding those models in Boolean formula, while Anastasakis et al. [42] advocate the use of Alloy for analyzing UML/OCL models and systematically study the transformation of those models to Alloy. In our work, DOC2SMT employs a set of syntax-oriented rules to translate well-formed constraint clauses in OCL to SMT-LIB. Since SMT-LIB natively supports more data-types and operations, the translation from OCL expressions to SMT-LIB is easier and more straightforward than that from OCL to Boolean formula. Unlike the general UML/OCL models that UML2Alloy needs to process, the constraint clauses generated by DOC2SMT from natural language involve only a limited subset of OCL constructs, which makes DOC2SMT's translation from OCL to SMT-LIB much less complex than the translation of UML/OCL to Alloy with UML2Alloy.

6. CONCLUSIONS

In this paper, we propose the DOC2SMT technique to generate strong functional constraints from natural language documentations for library methods. DOC2SMT was able to generate correct constraints for 309 public methods from 24 Java classes, and the average generating time is just 2.7 minutes for each correct constraint. The generated constraints also enabled symbolic-execution-based test generation to produce 51.2 times more new tests for 24 utility methods.

ACKNOWLEDGEMENTS

This research is supported by the National Natural Science Foundation of China (61972193), the Hong Kong RGC General Research Fund (GRF) under Grant PolyU 152002/18E, and the Fundamental Research Funds for the Central Universities of China (14380027).

REFERENCES

1. Ernst MD, Cockrell J, Griswold WG, Notkin D. Dynamically discovering likely program invariants to support program evolution. *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, ACM: New York, NY, USA, 1999; 213–224.
2. Csallner C, Tillmann N, Smaragdakis Y. Dysy: Dynamic symbolic execution for invariant inference. *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, ACM: New York, NY, USA, 2008; 281–290.
3. Nimmer JW, Ernst MD. Automatic generation of program specifications. *SIGSOFT Softw. Eng. Notes* Jul 2002; **27**(4):229–239.
4. Polikarpova N, Ciupa I, Meyer B. A comparative study of programmer-written and automatically inferred contracts. *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, ACM: New York, NY, USA, 2009; 93–104.
5. Wei Y, Furia CA, Kazmin N, Meyer B. Inferring better contracts. *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, ACM: New York, NY, USA, 2011; 191–200.
6. Ramanathan MK, Grama A, Jagannathan S. Static specification inference using predicate mining. *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, ACM: New York, NY, USA, 2007; 123–134.
7. Singleton JL, Leavens GT, Rajan H, Cok D. An algorithm and tool to infer practical postconditions. *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, ACM: New York, NY, USA, 2018; 313–314.
8. Pandita R, Xiao X, Zhong H, Xie T, Oney S, Paradkar A. Inferring method specifications from natural language api descriptions. *2012 34th International Conference on Software Engineering (ICSE)*, 2012; 815–825.
9. Zhai J, Huang J, Ma S, Zhang X, Tan L, Zhao J, Qin F. Automatic model generation from documentation for java api functions. *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016; 380–391.
10. Goffi A, Gorla A, Ernst MD, Pezzè M. Automatic generation of oracles for exceptional behaviors. *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, Association for Computing Machinery: New York, NY, USA, 2016; 213–224.
11. Blasi A, Goffi A, Kuznetsov K, Gorla A, Ernst MD, Pezzè M, Castellanos SD. Translating code comments to procedure specifications. *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, Association for Computing Machinery: New York, NY, USA, 2018; 242–253.

12. Manning CD, Surdeanu M, Bauer J, Finkel J, Bethard SJ, McClosky D. The Stanford CoreNLP natural language processing toolkit. *Association for Computational Linguistics (ACL) System Demonstrations*, 2014; 55–60.
13. Nguyen TH, Grundy J, Almorsy M. Rule-based extraction of goal-use case models from text. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, ACM: New York, NY, USA, 2015; 591–601.
14. de Moura L, Bjørner N. Z3-a tutorial 2011.
15. Barrett C, Fontaine P, Tinelli C. The SMT-LIB Standard: Version 2.6. *Technical Report*, Department of Computer Science, The University of Iowa 2017. Available at `www.SMT-LIB.org`.
16. Aho AV, Lam MS, Sethi R, Ullman JD. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley: Boston, MA, USA, 2006.
17. Jiang R, Chen Z, Zhang Z, Pei Y, Pan M, Zhang T. Semantics-based code search using input/output examples. *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2018; 92–102.
18. Meyer B, Ciupa I, Leitner A, Liu LL. Automatic testing of object-oriented software. *Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '07, Springer-Verlag: Berlin, Heidelberg, 2007; 114–129.
19. de Moura L, Bjørner N. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, Ramakrishnan CR, Rehof J (eds.), Springer Berlin Heidelberg: Berlin, Heidelberg, 2008; 337–340.
20. Steinberg D, Budinsky F, Paternostro M, Merks E. *EMF: Eclipse Modeling Framework 2.0*. 2nd edn., Addison-Wesley Professional, 2009.
21. Christian D, Adolfo SBH, Axel U, Edward W, contributors. Ocl documentation 2018.
22. Păsăreanu CS, Mehlitz PC, Bushnell DH, Gundy-Burlet K, Lowry M, Person S, Pape M. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, ACM: New York, NY, USA, 2008; 15–26.
23. Chen Z, Jiang R, Zhang Z, Pei Y, Pan M, Zhang T, Li X. Enhancing example-based code search with functional semantics. *Journal of Systems and Software* 2020; **165**:110 568.
24. Andoni A, Daniliuc D, Khurshid S, Marinov D. Evaluating the "small scope hypothesis" 10 2002; .
25. Zhou Y, Gu R, Chen T, Huang Z, Panichella S, Gall H. Analyzing apis documentation and code to detect directive defects. *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, IEEE Press, 2017; 27âĂŞ37, doi:10.1109/ICSE.2017.11. URL `https://doi.org/10.1109/ICSE.2017.11`.
26. Wen F, Nagy C, Bavota G, Lanza M. A large-scale empirical study on code-comment inconsistencies. *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, IEEE Press, 2019; 53âĂŞ64, doi:10.1109/ICPC.2019.00019. URL `https://doi.org/10.1109/ICPC.2019.00019`.
27. Zhou Y, Wang C, Yan X, Chen T, Panichella S, Gall H. Automatic detection and repair recommendation of directive defects in java api documentation. *IEEE Transactions on Software Engineering* 2020; **46**(9):1004–1023, doi:10.1109/TSE.2018.2872971.
28. Kremenek T, Twohey P, Back G, Ng A, Engler D. From uncertainty to belief: Inferring the specification within. *In Proceedings of the Seventh Symposium on Operating Systems Design and Implemetation*, USENIX Association, 2006; 161–176.
29. Nguyen HA, Dyer R, Nguyen TN, Rajan H. Mining preconditions of apis in large-scale code corpus. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, vol. 16-21-Nove, ACM Press: New York, New York, USA, 2014; 166–177.
30. Zhang L, Yang G, Rungta N, Person S, Khurshid S. Feedback-driven dynamic invariant discovery. *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, Association for Computing Machinery: New York, NY, USA, 2014; 362–372.
31. Nguyen T, Antonopoulos T, Ruef A, Hicks M. Counterexample-guided approach to finding numerical invariants. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, Association for Computing Machinery: New York, NY, USA, 2017; 605–615.
32. Li J, Sun J, Li L, Le QL, Lin SW. Automatic loop-invariant generation anc refinement through selective sampling. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017; 782–792.
33. Li K, Reichenbach C, Smaragdakis Y, Young M. Second-order constraints in dynamic invariant inference. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, Association for Computing Machinery: New York, NY, USA, 2013; 103–113.
34. Tan SH, Marinov D, Tan L, Leavens GT. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012; 260–269.
35. Wong E, Zhang L, Wang S, Liu T, Tan L. Dase: Document-assisted symbolic execution for improving automated software testing. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015; 620–631.
36. Sinha A, Jr SMS, Paradkar A. Text2test: Automated inspection of natural language use cases. *2010 Third International Conference on Software Testing, Verification and Validation*, 2010; 155–164.
37. Wang C, Pastore F, Goknil A, Briand L, Iqbal Z. Automatic generation of system test cases from use case specifications. *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, ACM: New York, NY, USA, 2015; 385–396.
38. Wang C, Pastore F, Briand L. Automated generation of constraints from use case specifications to support system testing. *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018; 23–33, doi:10.1109/ICST.2018.00013.
39. Bajwa IS, Bordbar B, Lee MG. Ocl constraints generation from natural language specification. *2010 14th IEEE International Enterprise Distributed Object Computing Conference*, 2010; 204–213, doi:10.1109/EDOC.2010.33.
40. Bajwa IS, Bordbar B, Anastasakis K, Lee M. On a chain of transformations for generating alloy from nl constraints. *Seventh International Conference on Digital Information Management (ICDIM 2012)*, 2012; 93–98, doi:10.1109/ICDIM.2012.6360153.

41. Soeken M, Wille R, Kuhlmann M, Gogolla M, Drechsler R. Verifying uml/ocl models using boolean satisfiability. *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, 2010; 1341–1344, doi:10.1109/DATE.2010.5457017.
42. Anastasakis K, Bordbar B, Georg G, Ray I. Uml2alloy: A challenging model transformation. *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems*, MODELS'07, Springer-Verlag: Berlin, Heidelberg, 2007; 436âĂŞ450.