Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2019-TR-001

2019-TR-001

Documentation-Based Functional Constraint Generation for Library Methods

Renhe Jiang, Minxue Pan, Yu Pei, Tian Zhang, Xuandong Li

Technical Report 2019

Documentation-Based Functional Constraint Generation for Library Methods

Renhe Jiang
Minxue Pan
131220105@smail.nju.edu.cn
mxp@nju.edu.cn
State Key Laboratory for Novel Software Technology, Nanjing University

Yu Pei
csypei@comp.polyu.edu.hk
Department of Computing, The Hong Kong Polytechnic University

Tian Zhang
Xuandong Li
ztluck@nju.edu.cn
lxd@nju.edu.cn
State Key Laboratory for Novel Software Technology, Nanjing University

ABSTRACT

Software libraries promote code reuse and facilitate software development, but they also increase the complexity of program analysis tasks. To effectively analyse programs built on top of software libraries, it is essential to have specifications that can be easily processed by analysis tools for the library methods. However, such specifications are absent currently: manual writing can be costly and error-prone, while existing automatic approaches do not provide specifications strong enough for 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 employs a novel "expansion and contraction" strategy. That is, it first explores many different ways to interpret the documentation for a method, and then determines, with help from a configurable domain model and a set of tests, which interpretation rightly captures the method's functionalities.

In experiments conducted on 259 methods from the Java Collections Framework, Doc2smt generated correct constraints for 180 methods. The average processing time is around 2 minutes per method. When the generated constraints are used to enable the symbolic execution of library methods in the Symbolic Java PathFinder, automated test generation produced 28.5 times more new tests for 16 utility methods.

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 facilitate common programming tasks and improve programmers' productivity. On the other hand, the libraries used in software development pose new challenges in the analysis of the resultant programs: the source code of the libraries may be hard to acquire, their binary code may be obfuscated, and they may be written in multiple programming languages and/or implement sophisticated engineering tricks for performance reasons.

Manual writing the specification for each method can be costly and error-prone. In view of that, researchers proposed various techniques in the past few years to automatically infer specifications for library methods, e.g., through dynamic [5, 8, 13, 15, 22] or static [17, 18] program analysis, so that their implementation details can be abstracted away to make complex program analysis tasks possible or scalable. Noticing that most program libraries provide concise yet well-summarised documentations for methods, one line of such work focuses on generating structured specifications for methods from their natural language written documentations [14, 23]. Pandita et al. [14] 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 descriptions 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. [23] construct code snippets as "model implementations" that are functionally equivalent to the methods under consideration. In their work, grammatical trees of sentences are transformed to produce variants, and pre-defined patterns are used to match tree structures and generate code snippets. These techniques extract useful information from library documentations. However, they do not produce concise and strong functional constraints, and therefore, cannot give full play to static analysis techniques: Alics mostly produces weak specifications such as null pointer assertions to facilitate the verification of legal usages; methods invoked in model implementations may still be challenging to automatically analyse. Besides, they rely heavily on heuristic translation rules. Considerable effort may be required to adapt the rules when applying the techniques to a wider range of libraries.

In response to the limitations, we propose a novel approach, named Doc2smt, to generating strong functional constraints for library methods based on their documentations. Instead of carefully customising the rules for processing the documentations, Doc2smt employs a set of general rules to translate descriptions of method functionalities into a large number of candidate OCL expressions. A novel two-step validation is then introduced to find out the constraints that comply with the behaviours of the method: In static validation, a domain model that can be manually enhanced incrementally is used to filter out OCL expressions that do not "fit" the problem domain; Dynamic validation checks whether a candidate constraint rightly abstracts the method under consideration through testing.

We implemented the approach into a tool also called Doc2smt. To evaluate the tool's effectiveness, we applied it to generate constraints for all the 259 public methods defined in 10 classes from the Java Collections Framework. Doc2smt successfully produced valid constraints for 182 methods, constraints for 180 methods among which are confirmed to be correct after manual inspection. The average time Doc2smt takes to process a method is around 2 minutes, and the amount of manual effort required to prepare all the inputs is

```
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 Javadoc for class `TreeMap`.

```
public V replace(K k, V v)
Description copied from interface: Map
Replaces the entry for the specified key only if it is
currently mapped to some value.
Parameters
k - key with which the specified value is associated
v - value to be associated with the specified key
Returns: the previous value associated with the specified
key, or null if there was no mapping for the key
```

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

moderate. When used to facilitate symbolic-execution-based generation of tests for 16 utility methods manipulating container objects, the constraints Doc2smt produces help to increase the number of generated new tests by 28.5 times.

This paper makes the following contributions:

- We propose an approach to generating functional constraints for library methods. The approach does not rely heavily on heuristical rules to process texts in natural language, and it is able to produce strong functional constraints in SMT that are readily usable by program analysis tools.
- We implement the approach into a prototype tool and conduct experiments to evaluate the approach. Experimental results suggest the approach is effective and efficient.

Tool Availability. A package with Doc2smt's implementation and all experimental data is publicly available at: *link removed for double-blind review.*

2 DOC2SMT IN ACTION

In this section, we use a method to demonstrate how Doc2smt generates strong functional constraint based on the corresponding 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 standard library implements a navigable map, i.e., a map whose elements can be easily accessed in ascending or descending key order, and it stores a pair of key and value as an `Entry` internally. Method `replace(K k, V v)` of the class, inherited from interface `Map`, substitutes the existing value to which `k` maps with `v`.

Developers of the class have written structured, highly informative documentation, in the form of Javadoc, to specify the class and its public methods. Figure 1 shows part of the Javadoc for

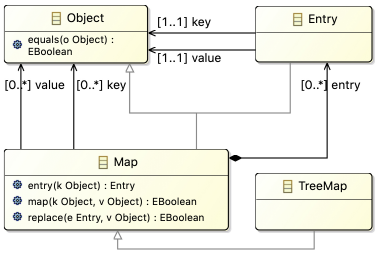

Figure 3: A UML class diagram as the (partial) domain model for class `TreeMap`. Public methods that can be easily extracted from the class Javadoc are omitted for brevity reasons.

```
1  ;declare-datatypes:
2  ;Map (mk-map (key (Array K Bool)) (mapping (Array K V)))
3  ;Entry (mk-entry (key K) (value V))
4  ;declare-const:
5  ;?p0 (Map Int Int), ?p1 Int, ?p2 Int,
6  ;?r Int, ?_p0 (Map Int Int), t0 (Entry Int Int)
7  ;assertions:
8  (= t0 ((as mk-entry) ?p1 (select (mapping ?p0) ?p1)))
9  (ite
10   (and (select (key ?p0) (key t0))
11     (= (select (mapping ?p0) (key t0)) (value t0)))
12   (and (= (key ?_p0) (key ?p0))
13     (= (mapping ?_p0) (store (mapping ?p0) (key t0) ?p2)))
14   (and (= (key ?_p0) (key ?p0))
15     (= (mapping ?_p0) (mapping ?p0))))
```

Figure 4: The SMT constraint generated by Doc2smt for method `TreeMap::replace(K k, V v)`. A `Map` object contains a `key` set to identify 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` correspond to the three input parameters of the method, while symbols `?_p0` and `?r` to the two output parameters.

```
Description
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
Returns
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 description and one from the return value description for `TreeMap::replace(K k, V v)`. The conjunction of expressions c) and d) captures the full functionality of the method.

the class, and Figure 2 shows the summary of its method `replace` from the same Javadoc file. Since the information is provided in natural language, it is not readily usable by, e.g., program analysis tools. Taking the documentation for the class and the domain model shown in Figure 3 as the input, Doc2smt generates in less than 5 minutes a SMT constraint, shown in Figure 4, that rightly captures the functionality of the method.

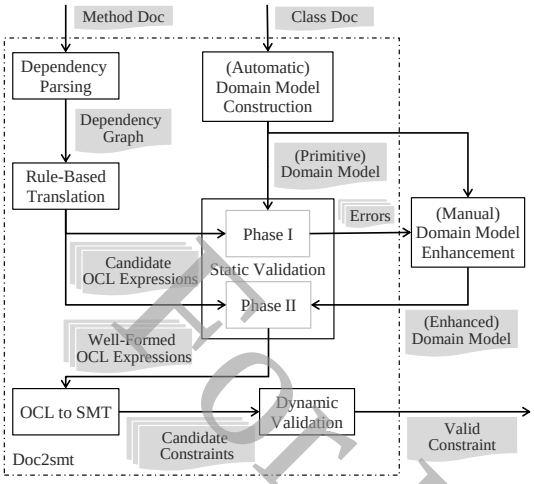

Figure 6: An overview of the Doc2smt approach.

During the process, Doc2smt generates a large number of OCL expressions as intermediate results. Figure 5 shows some of such expressions derived from the method description and the method return value specification. The SMT constraint is a direct translation of the OCL expression `c)and d)`.

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 general and 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 clearly do not apply to the specific problem domain. While manual work is needed in preparing the model, the amount of effort required is moderate and the tool provides guidance on the preparation. Third, Doc2smt runs tests to ensure only constraints that truly capture the dynamic behaviours of the methods are reported to the user.

3 THE DOC2SMT APPROACH

Figure 6 shows an overview of the Doc2smt approach. Inputs to Doc2smt include the documentation for a method and its defining class. Dependency parsing constructs a dependency graph from the method's summary (Section 3.1), and rule-based translation produces a rich set of candidate constraint clauses in the OCL syntax based on the dependency graph (Section 3.2). For phase I static validation, an automatically constructed, primitive domain model is used to help users identify domain knowledge necessary for characterising the method's functionality, while phase II static validation uses a manually enhanced domain model to prune out candidate clauses that refer to information outside the domain model (Section 3.3). The semantic equivalence between a constraint and the method implementation is then checked through testing during dynamic validation (Section 3.5). Doc2smt terminates upon discovering the first constraint that survives dynamic validation and outputs it as the one that captures the functionality of the method.

Given a method m defined in class C, let P and Q be the sequence of input and output parameters of m, respectively. Here the output

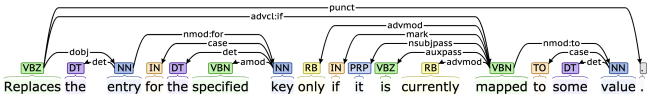

Figure 7: Dependency graph for the description of method `replace`.

parameters are for storing the return value of m, if any, and the values of the input parameters upon the return of m, if they are modified by m. The rest of this section details how Doc2smt generates, based on C's documentation, a functional constraint c for m.

3.1 Dependency Parsing of Method Summary

Doc2smt uses the summary of a method as the main source of information to learn about that method's functionality. The summary of a method includes the method signature, the name, type and description of each parameter, and a description of the method's functionality. If the method should return a value, the summary also includes the type and specification of that value. For example, the texts shown in Figure 2 constitute the summary of method `replace`.

To understand the functionality of a method, Doc2smt first employs a dependency parser to construct a dependency graph G_d from the method description. 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 of the description in Figure 2. According to the graph, word "replaces" is a verb in 3rd person singular present (VBZ), word "entry" is the direct object (dobj) of "replaces", and word "mapped" is an adverbial clause modifier (advcl) of "replaces".

We generate the graph to better understand the entities and their relationships in the sentences. Given its goal to generate functional constraints for methods, Doc2smt mainly focuses on POS tags concerning nouns (e.g., NNS for plural noun and NNP for singular proper noun) and verbs (e.g., VBD for past tense verb and VBN for past participle verb) in its analysis.

If the method returns a value, Doc2smt also constructs a dependency graph G_r based on the specification of that value. In the next step, Doc2smt translates G_d and G_r, if available, into sets of candidate constraint clauses in OCL.

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 OCL expressions, Doc2smt employs a rule-based technique.

3.2.1 Translation Rules. Doc2smt uses two types of rules in the translation: general rules and domain specific rules. Table 2 shows representative rules of each type. The syntax of rules is explained in Table 1.

General rules mainly focus on handling translations that are applicable to generic texts in English. For verbal phrases with simple structures, if the verbs are also names of operations in the problem domain, their translation is also processed by general rules. For

Table 1: 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 [10], while the syntax of actions is inspired by [12].

Nonterminal	Syntax	Explanation
`rule`	`pattern→action`	A translation rule consists of two parts: a `pattern` specifying 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`	Matches a node, referred to as `X`, with specific attribute values. **Example:** `{tag:NN.*}=A`.
`edgedec`	`>dep?(X,Y)=E`	Matches 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`	Matches 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`	Matches a graph where no edge matches with `edgedec`. **Example:** `not >(A,{})`.
`action`	`([grop]|plaintext)*`	An `action` consists of a sequence of graph `operations` (`grops`) and/or `plaintexts`. A `grop` specifies how parts identified in pattern matching are used during translation, while the `plaintexts` are directly copied into the translation results.
`grop`	`X`	Returns node X declared in the corresponding pattern.
	`{attrvalue*}`	Creates and returns a new node with the given attributes. **Example:** `{lemma:index}`.
	`copy(X)|deepcopy(X)`	Returns a copy/deep-copy of node `X`. **Example:** `copy(X)`
	$\texttt{grop}_1$`-`$\texttt{grop}_2$	Removes nodes and edges in $\texttt{grop}_2$ from $\texttt{grop}_1$ and returns $\texttt{gopt}_1$.
	`>E(`$\texttt{grop}_1$`,`$\texttt{grop}_2$`)`	Connects the root of $\texttt{grop}_2$ to that of $\texttt{grop}_1$ via edge `E` and returns $\texttt{grop}_1$. **Example:** `>E(X-Z,deepCopy(Y))`.
	`>dep(`$\texttt{grop}_1$`,`$\texttt{grop}_2$`)`	Connects the root of $\texttt{grop}_2$ to that of $\texttt{grop}_1$ via an edge of type `dep` and returns $\texttt{grop}_1$. **Example:** `>dobj(A,B)`
	`rep(X,`$\texttt{grop}_1$`,`$\texttt{grop}_2$`)`	Replaces node `X` in $\texttt{grop}_2$ with the result of $\texttt{grop}_1$ and returns $\texttt{grop}_2$. **Example:** `rep(Z,copy(Y),X)`.

Table 2: Example translation rules used in Doc2smt. For each category, its number of variants (#) and an example with id and definition. Categories on the top are general, while those on the bottom are domain specific.

CATEGORY	#	EXAMPLE	
		ID	DEFINITION
Pronoun Replacement	1	PR1	`{}=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)]`
Condition Translation	3	CT3	`{}=X {}=Y {}=Z 0,3>>conj:or(X,Y) 0,3>>(X,Z) >(Z,{lemma:if}) → if [Z] then [Y-Z] else [X-Y-Z] endif`
Quantifier Introduction	4	QI4	`{}=X {tag:NN.*}=Y {lemma:all|each}=Z 0,2>>(X,Y) >det.*(Y,Z)→[Y-Z]->forAll([copy(Y)]|[X-Z])`
Boolean Evaluation	7	BE7	`{lemma:greater|less}=X {tag:NN.*}=Y {tag:NN.*}=Z >nsubj(X,Y) >>nmod.*(X,Z) → [Y].[copy(X)]([Z],specifyinclusive)`
Passive to Active	1	PA1	`{tag:VBN}=X {tag:NN.*}=Y {tag:VB.*}=Z >nsubj.*(X,Y) >auxpass(X,Z) → [>dobj(X-Y-Z,Y)]`
Sentence Decompose	6	SD6	`{}=X {tag:NN.*}=Y {}=Z >(X,Y)=E >acl.*(Y,Z) → [Y] implies [>E(X-Y,copy(Y))]`
Noun Composition	1	NC1	`{tag:NN.*}=X {}=Y >amod|compound(X,Y) → [Y][X-Y]`
Noun Modifier	4	NM4	`{}=X {}=Y 0,3>>nmod.*(X,Y) → [X-Y] /→ [Y].[X-Y] /→ [X-Y]([Y]) /→ [Y]`
Verb Dobj	1	VD1	`{}=X {tag:NN.*}=Y >dobj(X,Y) → [X-Y]([Y])`
Verb Nsubj	1	VN1	`{}=X {tag:NN.*}=Y >nsubj(X,Y) → [Y].[X-Y]()`
Adjective Clause	2	AC2	`{tag:NN.*}=X {tag:VB.*}=Y >acl.*(X,Y) → [>nmod(Y,X-Y)]`
Terminal	1	TE1	`{}=X not >nmod.*(X,{}) → [copy(X)]`
This Reference	1	TR1	`{lemma:CN1|CN2|CN3}=X {lemma:this|the}=Y >det(X,Y) not >(X,{word:specified}) → self`
Special Structure	5	SS5	`{lemma:be}=X {lemma:there}=Y {tag:NN.*}=Z >expl(X,Y) >nsubj(X,Z) → contains([Z])`
Implicit Constraint	6	IC6	`{lemma:replace}=X {lemma:entry}=Y not >nmod.*(X,{}) → [X-Y]([Y],specifyvalue)`

example, Rule PR1, and Rule PA1, replaces pronouns with nouns they refer to, and changes a sentence in passive voice to active voice, respectively, while Rule VD1 translates a verbal phrase with direct object into a method invocation with argument.

Domain specific rules are complementary to general rules and devised to handle three types of scenarios that often occur in processing library documentations. First, when, e.g., a phrase "this treemap" appears in the summary of a method from class `TreeMap`, it 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 TR1 is introduced to deal with such cases particularly, where `CN`s are names of the context classes. Second, certain domain specific operations are often phrased in different ways in documentations, special rules are therefore needed to help identify those operations.

```
   Input: G: a dependency graph;
          Σ: a group 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().lemma} ;
4    else
5      Π ← ∅ ;
6      foreach σ ∈ Σ do
7        matcher ← σ.pattern.match(G) ;
8        while matcher.find() do
9          foreach action ∈ σ.actions do
10           Δ ← {action.toString()} ;
11           foreach grop ∈ action.getGrops() do
12             G′ ← grop.execute(G, matcher) ;
13             Π′ ← TRANSLATE(G′, Σ) ;
14             Δ ← ⋃_{δ∈Δ} ⋃_{π∈Π′} δ[grop/π] ;
15           end
16         end
17         Π ← Π ∪ Δ ;
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.

For example, Rule SS5 stipulates that a phrase `there is an o in c` can be translated into `c.contains(o)`. Third, constraints may be implied, instead of explicitly stated, in the documentation, and parts of a sentence may be omitted when the meaning is clear (for human) from the context. Domain specific rules are also needed in those cases to explicitly add the implicit or omitted information back during translation. Consider the method description in Figure 2. Verb `replace` is missing its complement "with sth", which, judging from the context, refers to parameter `v` of the method. To make the information complete in the translation, Rule IC6 includes a placeholder `specifyvalue` in its action to indicate that a specific value from the context should be used.

It is worth noting that, while the domain specific rules are closely related to the task of library documentation processing, they are general and most likely reusable across different libraries in the same language.

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 small number of options in making the translation, with rules that tend to manipulate a larger range of texts being attempted first. For instance, Rule CT3 will be tried before Rule PR1.

Given a dependency graph G and a set Σ of translation rules, function TRANSLATE shown in Figure 8 translates G into a set of OCL expressions in string by iterating through all possible ways to apply the rules in Σ. In case 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), repeatedly finds subgraphs in G that match the pattern specified in σ (line 8), and applies each action defined in σ to produce one translation (lines 9 through16). When applying an *action* to a matched subgraph, the function uses a set Δ to temporarily store the partial translation results produced by executing some of the graph operations defined in *action*: For each *grop* defined in *action* (line 11), the *grop* is first executed to produce the result graph G' (line 12), then G' is recursively translated to 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).

Consider again the method description in Figure 2 for example. Besides the three OCL expressions shown in Figure 5, Doc2smt also produces 324,336 other translations for it.

3.2.3 Post-Processing. Two extra general rules are employed to correct obvious syntax errors in translations.

Some translation rules (e.g. Rule BE7 and Rule SS5) refer to unspecified values from the context via placeholders like `specifyvalue`. Rule *Parameter Substitution* will replace such placeholders with actual values. For instance, `specifyvalue` will be replaced with the actual parameter `v` of the method when translating the description of `replace`.

Translation rules like NM4, VD1, and VN1 will surround each object of a verb with a pair of parentheses. If we regard the verb as denoting an operation, all the parameters, however, should be placed inside a single pair of parentheses and separated with commas. Rule *Parentheses Elimination* removes redundant parentheses and adds commas if necessary. For example, an invocation `map(key)(value)` would be changed to `map(key,value)` after parentheses elimination.

Applying rule-based translation to G_d produces a set E_d of candidate OCL expressions in string format. When the method returns a non-void value, another set E_r of candidates expressions is generated from G_r 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 can typically produce a large number of constraint clauses in OCL for a method. To determine whether a clause is well-formed or not, a context has to be provided. We propose a domain model as part of the input for that purpose, and a two-step approach to preparing the model incrementally.

Since all the related classes and their methods are clearly essential for the problem domain under consideration, and the information is easily retrievable in the corresponding class documentation, Doc2smt automatically extracts the information and builds a primitive domain model based on that. For example, the primitive domain model extracted for class `TreeMap` will include types like `Object` and `TreeMap` and methods like `containsKey`, `get`, and `put`.

While this primitive model contains important information about the problem domain, it does not show how classes are related. More importantly, the description of class and method functionalities in a Javadoc may involve (high-level) implementation details, which should be reflected in the domain model to help validate the OCL expressions extracted from the same Javadoc. In the example with `TreeMap`, since `Entry` is a public nested class of `TreeMap`, it is also part

of the primitive model Doc2smt constructs for `TreeMap`. Method `replace` refers to "the entry for the specified key" in its description, but no operation is provided by `TreeMap` to get the entry object using a key. Doc2smt relies on user input to complement the primitive domain model with operations like `entry(Object)`.

Since our ultimate goal is to generate functional constraints for methods, we focus on parts of the domain model that will facilitate such generation. Particularly, Doc2smt first validates all the generated OCL expressions against the primitive domain model. Besides of identifying expressions with syntax errors, Doc2smt also records validation errors due to missing entities (i.e. classes and/or properties) in the model, and reports them to the user in descending order of their counts of occurrences. In *manual* domain model enhancement, a user may choose to focus only on the missing entities with counts of occurrences greater than N_o: Only relevant entities should be added to the model, while the others should be ignored.

Programmers often use synonyms in documentations to avoid repeating the same words. Such synonyms can be one of the causes for missing entities. For example, a reference to a missing property named `beginning` can be replaced with that to an existing property named `start` in the same context, since "beginning" and "start" are synonyms. When checking through all the missing entities in this step, Doc2smt allows users to map the words with same semantics in a context, and creates a table of synonyms. It then uses the table to replace the words with their synonyms in generated OCL expressions.

Once we have the manually enhanced domain model, another round of static validation is conducted to prune out OCL expressions that fail to validate. For instance, based on the enhanced model given in Figure 3, clause a) in Figure 5 is considered invalid since it refers to a property `k` of `value`, which, however, does not exist according to the domain model. Constraint clauses that cause no errors in either phase of static validation are referred to as *well-formed* clauses.

Let W_d and W_r be the set of well-formed clauses from E_d and E_r ($W_d \subseteq E_d, W_r \subseteq E_r$), respectively. Since a functional constraint is expected to incorporate all the requirements from both the method description and the method return value specification, Doc2smt computes a set $C = \{e_1 \texttt{ and } e_2 \mid e_1 \in W_r, e_2 \in W_d\}$ as the set of candidate constraints for the method. Each candidate constraint $\sigma \in C$ is a predicate on P and Q.

Note that while the well-formed clauses are in the OCL form, they may not adhere to the OCL standard specification, since we allow 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 choose to have constraint clauses in the OCL form at this step to enable, with the help of an OCL expression validator, the easy identification of valid relations among properties, operations, and objects in the problem domain and the effective prune of most invalid relations suggested by the permissive translation process Ocl2smt implements.

```
1 entry(Object)
2 (= ?r ((as mk-entry) ?p1 (select (mapping ?p0) ?p1)))
3 contains(Entry)
4 (= ?r (and (select (key ?p0) (key ?p1))
5     (= (select (mapping ?p0) (key ?p1)) (value ?p1))))
```

Figure 9: SMT constraints for meta-operations `entry` and `contains` from class `TreeMap`.

3.4 OCL to SMT

Candidate constraints are then translated to the SMT-LIB [3, 7] format via a syntax-directed process [1].

In this step, expressions like `if-then-else-endif` in OCL are directly translated to expressions like `ite` in SMT-LIB, but operations like `replace` and `entry` in Figure 5.c need special treatments, since there are no constraints expressing their semantics. We refer to such operations as meta-operations. Doc2smt automatically identifies meta-operations and demands their semantics to be provided in SMT in the enhanced domain model (one by one or in batch).

If one operation is used to express the constraint of another operation, Doc2smt incorporates the semantics of the former into that of the latter using a similar technique as applied in [9]. Consider an operation f invoked in a candidate constraint, let $\mathcal{S}$ be the encoding of f's semantics in SMT-LIB. Essentially, to translate an invocation to f to SMT-LIB, Doc2smt first instantiates $\mathcal{S}$ using a unique variable for each input and output formal parameter of f, then conjuncts that instantiation to the existing translation result, and finally binds actual parameters with those unique variables corresponding to the formal parameters.

While quantifications in OCL can be easily translated into their counter-constructs in SMT-LIB, to help make the translation results easier to solve, we can provide rules in the enhanced domain model to guide the elimination of quantifiers. For example, one such rule may suggest to replace the existential quantifier `value->exists(value|map(k,value))` in Figure 5.c with the call sequence `contains(entry(k))`. Figure 9 shows the SMT constraints for meta-operations `entry` and `contains`. The instantiations of the two constraints appear on lines 8, 10, and 11 in Figure 4.

3.5 Dynamic Validation

For a constraint c to be a *correct*, i.e., both complete and sound, functional constraint for m, c has to satisfy the condition that, given two sequences of values p and q that are compatible with P and Q, respectively, i) $c(p, q)$ holds *if and only if* ii) invoking method m on input p produces output q.

It is obviously impossible to examine *all* feasible input and output values of m in most cases, and therefore, Doc2smt checks whether two conditions i) and ii) are equivalent, with respect to a limited set of tests. Particularly,

- Doc2smt implements a random algorithm [11] to generate a group T_1 of N_t tests for m. If for every test $t \in T_1$, the input values p_1 and output values q_1 of t satisfy c, i.e., $c(p_1, q_1)$, then condition i) is *necessary* for condition ii), w.r.t. T_1;
- Doc2smt also utilises the off-the-shelf constraint solver Z3 [6] to find a set S of N_s solutions for each candidate constraint. Let p_s and q_s be the input and output values of a solution $s \in S$, t_s be the test for m with p_s as the input

Table 3: Subjects used in experiments to answer RQ3. For each context CLASS, the number of subject methods selected from that class (#METHOD) and the types of containers the subject methods manipulate (CONTAINER CLASSES).

CLASS	#METHOD	CONTAINER CLASSES
`Collections`	8	`Collection`, `List`
`CollectionUtils`†	2	`Collection`, `Set`, `HashSet`
`ListUtils`†	5	`Collection`, `Set`, `HashSet`, `List`, `ArrayList`, `Map`, `HashMap`
`MapUtils`†	1	`Map`
TOTAL	16	-

† Classes from package `org.apache.commons.collections4`.
Class `Collections` is from package `java.util`.

parameters, and $T_2 = \cup_{s \in S} t_s$ be the set of all tests derived from S. If for every solution $s \in S$, the output of t_s is equal to q_s, then condition i) is *sufficient* for condition ii), w.r.t. T_2.

If condition i) is both necessary and sufficient for condition ii), w.r.t. the set $T = T_1 \cup T_2$ of tests, we regard c as valid, w.r.t. T.

Dynamic validation stops once a valid constraint is found; that constraint is returned by DOC2SMT as the function constraint for m. No constraint will be returned if no valid one is found.

3.6 Implementation Details

We have implemented the technique described above into a tool, also named DOC2SMT. The tool integrates the Stanford CoreNLP Toolkit [10] 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 [4] is used to statically validate constraint clauses and translate candidate constraints into SMT-LIB format.

DOC2SMT, however, 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 analyse for how many methods DOC2SMT is able to generate constraints and what quality the constraints have.
- **RQ2:** How efficient is DOC2SMT? In RQ2, we focus on the cost of applying DOC2SMT to generate functional constraints, and the breakdown of the overall time cost into the time for candidate clause generation, static validation, and dynamic validation.
- **RQ3:** How useful are the generated constraints? In RQ3, we assess the usefulness of the constraints in terms of the improvements they bring to symbolic-execution-based test generation.

4.1 Subjects

To answer RQ1 and RQ2, we choose 10 common container classes from the Java Collections Framework[1] (JCF), since container classes are notorious for their complexity that affects program analysis,

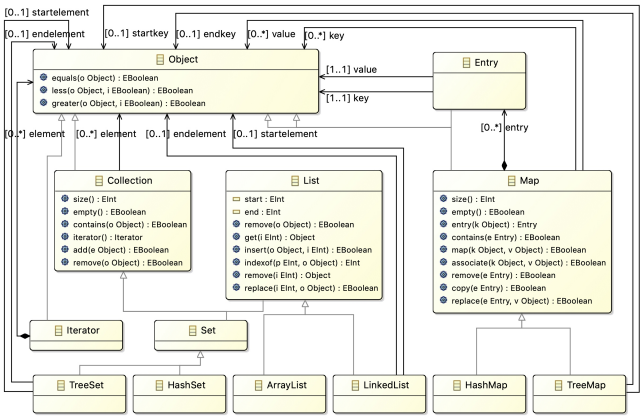

Figure 10: Enhanced domain model used in the experiments. Note that most public methods listed in the documentations are omitted here for brevity reasons. Also note that `EInt` and `EBoolean` are counter-types of `int` and `boolean` in EMF.

and yet they are among the most widely used libraries. All the methods described in the documentation of container classes are used as the subjects in our experiments. Columns CLASS and #M of Table 4 show the subject classes and the number of subject methods chosen from each class.

To answer RQ3, we searched through 4 classes from JCF and the Apache Commons Collections[2] (ACC) for static methods manipulating variables of primitive types and methods of the 10 container classes. Targeting such methods is simply to ease the running of SPF, since it could be non-trivial to run SPF depending on the characteristic of methods. Static methods are preferred by SPF, as they do not require the instantiations of objects, which may require specific values. Same are the methods only containing primitive variables and invoking methods with functional constraints. We gather in total 16 such methods as our subjects. Table 3 gives information about the context classes of the methods and the types of container objects the methods manipulate.

4.2 Experimental Protocol

Our experiments are divided into two parts. We apply DOC2SMT to generate constraints for methods from the 10 container classes in the first part, and use the generated constraints to produce test suites for 16 utility methods in the second part.

In the first part of experiments, for each subject method, we first use a simple script to extract the summary from Javadoc documentation, then feed it to DOC2SMT. DOC2SMT translates the summary into constraint clauses in OCL syntax. During domain model enhancement, we select the missing entities reported by DOC2SMT with counts of occurrences greater than $N_o = 10$, and only add the relevant ones to the domain model. As the result, we construct the enhanced domain model, as shown in Figure 10, for the classes. Note that, for brevity reasons, the model only shows entities that are extra to the public methods specified in class documentations or meta-operations. Public methods listed in the class documentations

[1] https://docs.oracle.com/javase/8/docs/technotes/guides/collections/

[2] https://commons.apache.org/proper/commons-collections/

Table 4: For each CLASS from package `java.util`, the numbers of methods used as the subject (#M), valid (#V) and correct (#C) CONSTRAINTS produced, and candidate constraint clauses GENerated (#L); the numbers of well-formed constraint clauses retained (#WF-L) and methods with well-formed constraints (#M') after static validation; the numbers of well-formed constraints checked until finding the first valid (#WF-C), solutions for SMT constraints generated by Z3 (#SOLU), and tests produced (#TEST) during DYNAMIC validation; the overall running time of DOC2SMT (T), the time spent on candidate constraint clause generation (T_G), static validation (T_{SV}), and dynamic validation (T_{DV}). All times are in seconds.

		CONSTRAINT		GEN	S-V		D-V			TIME			
CLASS	#M	#V	#C	#L	#WF-L	#M'	#WF-C	#SOLU	#TEST	T	T_G	T_{SV}	T_{DV}
`Collection`	19	9	9	604K	160	13	32	644	900	2253.57	0.69	3.45	2246.67
`Set`	16	11	11	2737K	142	13	25	812	1100	2801.97	0.70	112.99	2576.00
`HashSet`	9	8	8	65K	37	8	8	530	800	1354.21	0.31	5.97	1342.26
`TreeSet`	27	24	23	4831K	192	25	34	1884	2400	5275.49	1.67	380.04	4515.41
`List`	28	17	17	13901K	206	21	28	704	1418	1720.41	1.60	376.99	966.43
`ArrayList`	31	16	16	14415K	234	20	27	758	1362	1799.77	1.16	376.95	1045.88
`LinkedList`	40	34	33	13864K	483	37	48	1473	3152	2538.53	1.67	297.81	1942.90
`Map`	25	15	15	10077K	222	18	21	908	1439	3263.13	10.07	425.82	2411.49
`HashMap`	24	15	15	10141K	213	18	23	839	1445	4019.82	9.20	410.13	3199.56
`TreeMap`	40	33	33	8506K	288	35	39	2223	3245	7325.49	3.65	948.28	5428.94
Total	259	182	180	79141K	2177	208	285	10775	17261	32352.39	30.72	3338.43	25675.54

are omitted in the figure. Each phase of static validation is configured to run for at most 2 minutes. In dynamic validation, DOC2SMT is configured to generate at most $N_t = 100$ different tests for each method and find at most $N_s = 100$ solutions for each candidate constraint.

We decide whether a valid constraint is correct or not through manual inspection. We are aware that manual assessment may cause a major threat to the construct validity of our findings. To mitigate the threat, two authors independently examine the quality of each valid constraint reported by DOC2SMT. A constraint is only marked as *correct* if both authors agree that the constraint properly captures all the functionalities of the corresponding method. We leave a detailed analysis of constraint correctness for future work.

Besides whether a valid constraint is produced and whether that constraint is indeed correct, we also record the following measures for the run of DOC2SMT on each method:

- #L: number of candidate constraint clauses in OCL generated;
- #WF-L: number of well-formed constraint clauses in OCL retained after static validation;
- #WF-C: number of well-formed candidate constraints dynamically validated until a valid one is found;
- #SOLU: number of solutions the solver produces for all the candidate constraints;
- #TEST: number of tests generated for all the methods;
- 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;

In the second part of experiments, we use generated constraints to enhance the capability of Symbolic Pathfinder (SPF) [16] in test suite generation. SPF is a symbolic execution engine for Java, built on the top of the Java PathFinder (JPF) model checker. It introduces a customised bytecode instruction factory to augment the concrete execution semantics of Java programs with symbolic execution, and it attaches a field `attribute` to each variable for storing the symbolic value of the variable. The enhanced SPF (ESPF) modifies the interpretation of `invoke` instructions in Java programs in such a way that, when an invocation to a method with functional constraint in SMT is encountered, it incorporates the constraint in the same way as is done during the translation from OCL to SMT (Section 3.4).

To generate tests for a utility method, we launch SPF/ESPF with an initial test for that method. Once entering the method body, SPF/ESPF exhaustively examines all possible execution paths of the method and reports the corresponding path conditions one by one. We then send each path condition to the Z3 solver. If the solver can find a solution to the condition and a test can actually be constructed using the solution to cover a new path, a new test has been generated. In this way, we are able to produce a group of new tests, each covering a distinct path than the initial test does. In this part of the experiments, SPF and ESPF are both configured to run on each method until either 100 different tests are generated or the 2-minute time limit is reached, and we record the number of new tests generated. Our choice of such completion criteria is motivated by the "small-scope hypothesis", which claims that many defects can be triggered with small inputs and witnessed using short executions [2].

Our experiments were conducted on a desktop computer running Ubuntu 16.04 on a Intel Core i7-6700 CPU (3.4GHz) and 16G RAM. The time out for each invocation to Z3 is set to 20 seconds.

4.3 Experimental Results

In this section, we report the experimental results as answers to the three research questions.

4.3.1 RQ1: Effectiveness. Table 4 shows that, in total, DOC2SMT was able to generate valid constraints for 182 of the 259 subject methods, achieving an overall success rate of 70%, which suggests DOC2SMT is highly applicable. Among the 182 valid constraints, 180 are indeed correct. The overall high quality of the results indicates that DOC2SMT is effective in generating strong functional constraints. Figure 11 depicts the distribution of contributions by different translation rule categories in the experiments. It is clear from the figure that each rule category is needed to successfully handle a significant number of methods.

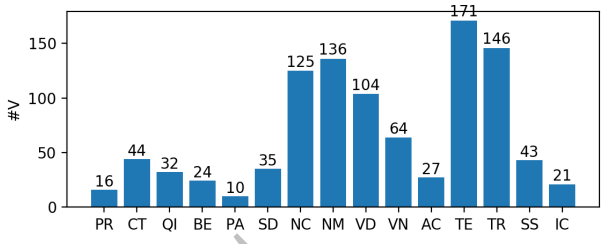

Figure 11: For each translation rule category, the number of valid constraints whose generation involves at least one application of a rule from that category.

The valid constraints for methods `TreeSet::descendingSet` and `LinkedList::pollLast`, however, were actually incorrect. The first method should return a navigable set of the same elements in the treeset but with a reversed navigation order. Since we did not provide the meta-operation to test the quality of navigable set objects properly, dynamic validation had to make do with the more general comparison defined for set objects. The criteria for valid constraints is therefore weakened, allowing the incorrect constraint to survive the dynamic validation. The second 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 is further down the list of all candidates, compared with the one Doc2smt returns, and is therefore missed by the tool.

Doc2smt was not able to produce any valid postcondition for 75 methods. We manually check these methods and identify four reasons for the failures: 1) Doc2smt failed to produce any well-formed constraint clause for 51 methods due to *incomplete domain model*. For example, since type `Comparator` is not part of the domain model, Doc2smt cannot generate any clause for method `TreeMap::comparator`, which returns an object of type `Comparator`, that passes the static validation; 2) For 6 other methods, while Doc2smt was able to generate the correct candidate constraints, the constraints were pruned out since 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 it; 3) Due to *limitations in the linguistic analysis*, Doc2smt was not able to properly handle the summaries and produced only invalid constraints for 7 methods; 4) Part of the semantics was only implied, instead of explicitly provided, in the documentations for 11 methods. Therefore Doc2smt failed to capture those constraints and produced only partial constraints for the methods, which were then invalidated during dynamic validation. For example, the documentation for method `Set::toarray` requires that the result contains all elements in the set, but not that the result should only contain elements from the set. The generated constraint reflects only the weak specification given in the method summary and admits solutions that do not comply with the actual dynamic behaviors of the method.

Table 5: Numbers of additional classes (#cls), operations (#op), and properties (#pr) in the enhanced domain model, compared with the primitive domain model.

#cls	#op	#pr	total
3	24	13	40

As listed in Table 4, Doc2smt generated, for all the subject methods, nearly 80 million candidate constraint clauses, among which only 2177 validated successfully with the domain model and are well-formed, leading to 285 well-formed constraints checked on 208 methods and 182 valid constraints produced. These numbers suggest that Doc2smt explores a fairly large space in constructing candidate constraint clauses, and its remaining steps are effective in pruning the invalid constraints and reporting only the ones of high quality.

4.3.2 RQ2: Efficiency. Since the generation process is not completely automated, we examine the efficiency of Doc2smt from two different aspects: the time cost for running the Doc2smt tool and the amount of manual effort required to prepare the inputs.

Table 4 also shows the time it takes for Doc2smt to produce the valid constraints and the breakdown of that into the time spent on each of the three main steps, i.e., candidate constraint generation, static validation, and dynamic validation. In total, it took Doc2smt 539 minutes to produce all the valid constraints, averaging to 53.9 minutes for each class or 2 minutes for each method. Among the three main steps, dynamic validation is by far the most time-consuming, which is understandable given its nature of dynamic analysis.

Manual effort needed in the experiments involves creating the 12 domain specific translation rules, enhancing the primitive domain model with 40 extra elements, and crafting SMT constraints encoding the semantics of 26 meta-operations in domain models. In phase I static validation, Doc2smt reports 261 errors due to missing entities in total. Since most errors occurred less than 10 times, we only had to check 44 missing entities manually. Table 5 lists the numbers of additional classes, operations, and properties added to enhance the primitive domain model.

In view that Doc2smt generated correct constraints for 180 methods from 10 classes, and many of the domain specific translation rules can be reused to process other Java library methods, we consider the amount of manual effort involved in applying Doc2smt as moderate. We leave a more systematic and quantitative investigation into this aspect for future work.

4.3.3 RQ3: Usefulness in test generation. In total, espf generated 683 new tests for the 16 utility methods, while spf was only able to generate 24 new tests for 3 methods. Table 6 shows that spf was not able to generate any new test on 13 methods. The reason is that spf will use concrete, instead of symbolic, values when encountering objects it cannot generate symbolic expressions for, e.g. a container symbolic input. Under such a circumstance, no path condition will be constructed from the execution, therefore no new inputs can be generated to drive the execution to explore a different path. For ESPF, we increase its capability to interpret container objects and method invocations, to explore more feasible paths.

Table 6: Comparison between SPF and ESPF in test suite generation. For each tool and each method, the number of *new* tests generated and the time spent to achieve that in Milliseconds.

	ESPF		SPF	
METHOD	T	MS	T	MS
`Collections.indexedBinarySearch`	15	659	6	120
`Collections.reverse`	2	175	0	71
`Collections.max`	9	4741	0	68
`Collections.rotate`[1]	12	33352	9	634
`Collections.rotate`[2]	13	25084	9	233
`Collections.indexOfSubList`	98	4442	0	42
`Collections.lastIndexOfSubList`	93	9094	0	41
`Collections.disjoint`	19	3070	0	42
`CollectionUtils.containsAll`	96	120084	0	72
`CollectionUtils.containsAny`	17	2388	0	81
`ListUtils.intersection`	7	98181	0	89
`ListUtils.subtract`	99	10347	0	52
`ListUtils.union`	0	845	0	54
`ListUtils.retainAll`	99	45248	0	40
`ListUtils.removeAll`	99	24428	0	52
`MapUtils.invertMap`	5	3782	0	49
Total	683	385920	24	1740

Regarding the execution time, it is understandable that SPF executes much faster than ESPF, since little symbolic execution, and analysis in consequence, can be done there. Such experimental results suggest that the constraints generated by Doc2SMT can be used to effectively improve test generation.

4.4 Limitations

While preconditions and exceptions that might be raised during its execution are also important information regarding a method's interface, and useful for analyzing code that invokes the method, we focus on generating strong functional constraints that capture the postconditions of library methods in this work. One interesting direction we plan to explore in the future is to advance the Doc2SMT approach to extract also method preconditions and class invariants from documentations.

In this work, we evaluate the effectiveness and efficiency of Doc2SMT on public methods defined in ten container classes from the Java Collections Framework. While experimental results show that Doc2SMT is reasonably effective and efficient on the methods, all these classes implement popular data structures with well-defined interfaces, and they may not be a good representative of libraries in other areas and/or languages. In the future, we will conduct larger scale experiments on more methods from various libraries to understand better the applicability of our approach.

5 RELATED WORK

The work of this paper is based on results from multiple research areas. For space reasons, this section briefly reviews researches in specification inference and model-driven engineering that have the largest influences on this work.

Various techniques have been proposed to infer specifications for programs at various levels. Dynamic invariant detection [5, 8] infers invariants by dynamically running a program and using machine learning algorithms to analyze the execution traces, while other approaches try to improve specification quality by combining dynamic analysis with static analysis [13] or by exploiting other information, e.g., programmer-written contracts, in programs [15]. Many works have also been done to facilitate specification generation by analyzing documentations written in natural language. ALICS [14] is the first approach that analyses API documents to generate code contracts. Its text analysis engine translates sentences in API descriptions into first-order logical expressions based on pre-defined shallow parsing semantic templates [14], which are are analogous to *patterns* in our translation rules, and the expressions are then translated to code contracts based on a predefined mapping from predicates to programming constructs. ALICS does not aim to produce strong functional constraints capturing the full semantics of APIs, while Doc2SMT tries to generate specifications that are as complete as possible. Zhai et al. [23] 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. The text analysis engine they use generates a grammatical tree for a sentence, transforms such a tree to produce variants, and uses pre-defined patterns to match tree structure and generate code snippets. Zhou et al. [24] extracts constraints about exceptions both from source code and documentations to detect defects of API directives. Compared with these techniques, Doc2SMT works on dependency graphs and general translation rules, and it uses a domain model to manage domain knowledge and validate generated constraint clauses. To the best of our knowledge, Doc2SMT is the first NLP-based approach to generating specifications that are good enough to be utilised by techniques like symbolic execution, which hold a high expectation for the soundness and completeness of its input specifications.

The idea behind this paper is inspired by the work that uses model-driven techniques to facilitate the documentation analysis. Text2Test [19] is an approach to building models from use case specifications, and it facilitates engineers to revise their use cases based on the construction and analysis of models. The UMTG [21] approach generates system test cases from use case specifications. It first combines techniques in natural language processing and domain models to generate an use case test model from a specification, then derives test cases from the generated model. GUEST [12] is a rule-based approach to extract goal and use case models from natural language requirements documents. The idea of using a domain model to manage domain knowledge in this paper is inspired by UMTG, and the design of translation rule in this paper is inspired by the goal extraction rules proposed in GUEST.

6 CONCLUSIONS

Program specifications are important assistances to simplify API analysis tasks instead of struggling with code. Library documentation can be regarded as a high-level specification but is hard for computers to understand. In this paper, we propose the Doc2SMT technique to generate formal, functional constraints from natural language documentations for library methods, based on a library domain model, a divide and conquer algorithm, and a group of translation rules. Our experimental results show that Doc2SMT

generates correct constraints for 180 collection APIs among all 259 subjects, and the average generating time is about 2 minutes. The generated constraints increase the number of new tests produced by SPF on 16 realistic collection manipulating methods by 28.5 times, which confirms the efficacy of our work.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, Boston, MA, USA.

[2] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. 2002. Evaluating the "Small Scope Hypothesis". (10 2002).

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6*. Technical Report. Department of Computer Science, The University of Iowa. Available at www.SMT-LIB.org.

[4] Damus Christian, SÃąnchez-Barbudo H, Adolfo, Uhl Axel, Willink Edward, and contributors. 2018. OCL Documentation.

[5] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: Dynamic Symbolic Execution for Invariant Inference. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 281–290.

[6] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[7] Leonardo de Moura and Nikolaj Bjørner. 2011. Z3-a Tutorial.

[8] 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 (ICSE '99)*. ACM, New York, NY, USA, 213–224.

[9] R. Jiang, Z. Chen, Z. Zhang, Y. Pei, M. Pan, and T. Zhang. 2018. Semantics-Based Code Search Using Input/Output Examples. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 92–102. https://doi.org/10.1109/SCAM.2018.00018

[10] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*. 55–60.

[11] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Ling Liu. 2007. Automatic Testing of Object-Oriented Software. In *Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM '07)*. Springer-Verlag, Berlin, Heidelberg, 114–129.

[12] Tuong Huan Nguyen, John Grundy, and Mohamed Almorsy. 2015. Rule-based Extraction of Goal-use Case Models from Text. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 591–601. https://doi.org/10.1145/2786805.2786876

[13] Jeremy W. Nimmer and Michael D. Ernst. 2002. Automatic Generation of Program Specifications. *SIGSOFT Softw. Eng. Notes* 27, 4 (July 2002), 229–239. https://doi.org/10.1145/566171.566213

[14] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. 2012. Inferring method specifications from natural language API descriptions. In *2012 34th International Conference on Software Engineering (ICSE)*. 815–825. https://doi.org/10.1109/ICSE.2012.6227137

[15] Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. 2009. A Comparative Study of Programmer-written and Automatically Inferred Contracts. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA '09)*. ACM, New York, NY, USA, 93–104. https://doi.org/10.1145/1572272.1572284

[16] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. 2008. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing Nasa Software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/1390630.1390635

[17] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Static Specification Inference Using Predicate Mining. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 123–134.

[18] John L. Singleton, Gary T. Leavens, Hridesh Rajan, and David Cok. 2018. An Algorithm and Tool to Infer Practical Postconditions. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*. ACM, New York, NY, USA, 313–314.

[19] A. Sinha, S. M. S. Jr., and A. Paradkar. 2010. Text2Test: Automated Inspection of Natural Language Use Cases. In *2010 Third International Conference on Software Testing, Verification and Validation*. 155–164. https://doi.org/10.1109/ICST.2010.19

[20] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework 2.0* (2nd ed.). Addison-Wesley Professional.

[21] Chunhui Wang, Fabrizio Pastore, Arda Goknil, Lionel Briand, and Zohaib Iqbal. 2015. Automatic Generation of System Test Cases from Use Case Specifications. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 385–396. https://doi.org/10.1145/2771783.2771812

[22] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. 2011. Inferring Better Contracts. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 191–200.

[23] J. Zhai, J. Huang, S. Ma, X. Zhang, L. Tan, J. Zhao, and F. Qin. 2016. Automatic Model Generation from Documentation for Java API Functions. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 380–391. https://doi.org/10.1145/2884781.2884881

[24] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall. 2017. Analyzing APIs Documentation and Code to Detect Directive Defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 27–37. https://doi.org/10.1109/ICSE.2017.11