ABSTRACT

Code comments provide abundant information that have been leveraged to help perform various software engineering tasks, such as bug detection, specification inference, and code synthesis. However, developers are less motivated to write and update comments, making it infeasible and error-prone to leverage comments to facilitate software engineering tasks. In this paper, we propose to leverage program analysis to systematically derive, refine, and propagate comments. For example, by propagation via program analysis, comments can be passed on to code entities that are not commented such that code bugs can be detected leveraging the propagated comments. Developers usually comment on different aspects of code elements like methods, and use comments to describe various contents, such as functionalities and properties. To more effectively utilize comments, a fine-grained and elaborated taxonomy of comments and a reliable classifier to automatically categorize a comment are needed. In this paper, we build a comprehensive taxonomy and propose using program analysis to propagate comments. We develop a prototype CPC, and evaluate it on 5 projects. The evaluation results demonstrate 41573 new comments can be derived by propagation from other code locations with 88% accuracy. Among them, we can derive precise functional comments for 87 native methods that have neither existing comments nor source code. Leveraging the propagated comments, we detect 37 new bugs in open source large projects, 30 of which have been confirmed and fixed by developers, and 304 defects in existing comments (by looking at inconsistencies between existing and propagated comments), including 12 incomplete comments and 292 wrong comments. This demonstrates the effectiveness of our approach. Our user study confirms propagated comments align well with existing comments in terms of quality.
through program analysis, comments can be passed on to code entities that are not commented such that code bugs can be detected by cross-checking code with the propagated comments.

Due to the lack of standard of composing documentation, developers have substantial flexibility and they tend to use arbitrary ways to compose documentation. They usually comment on different aspects of code elements like classes, methods, and variables, and use comments to describe various contents, such as summarizing the functionality, explaining the design rationale and specifying the implementation details. In addition, as comments are written in natural language, they are intrinsically ambiguous and accurate linguistic analysis is needed to acquire their exact meanings and scopes. To better understand code and more effectively propagate comments, we must first know which code elements they comment on and what kind of information they convey. That is to say, it is imperative to design a fine-grained and elaborated taxonomy of code comments and develop a reliable classifier to automatically categorize a comment.

There have been efforts in software documentation classification. Padioleau et al. [58] built a taxonomy based on meanings of comments. The work in [46] proposed a taxonomy of knowledge types in API reference documentation and used the taxonomy to assess the knowledge they contain. Based on this taxonomy, researchers in [43] developed a set of textual features to automatically categorize the knowledge. In [74], researchers studied comment categorization to provide better quantitative insights for comment quality assessment. Features are manually given for machine learning techniques to automatically classify comments. Researchers in [60, 61] first manually classified more than 2,000 code comments and then used supervised learning to achieve about 85% classification accuracy. Their taxonomies are not designed to be coupled with program analysis. It is unclear how to propagate and infer comments based on their classification.

Hence, our goal is to first build a comprehensive taxonomy from different perspectives (e.g., what and why) and different code entities (e.g., class and method), and then design a uniform analysis to enable bi-directional analysis: (1) program analysis propagates and updates comments, and (2) comments provide additional semantic hints to enrich program analysis. Multiple software tasks can benefit from the bi-directional analysis. For example, leveraging program analysis to propagate comments can provide automation support in maintaining documentation which is difficult [21] and leveraging comment analysis can help detect software bugs by checking the comment semantics against source code.

In this paper, we propose CPC, a principled and sophisticated software reasoning method that couples comment analysis and program analysis. It automatically classifies comments based on different perspectives and code entities (namely builds a comment taxonomy), and thus each comment is attributed to a code element and becomes a first-class object just like other classic objects in program analysis (e.g., variables and statements). Based on the taxonomy, CPC leverages program analysis techniques to propagate comments from one code entity to another to update, infer, and associate comments with code entities. Then CPC extracts semantics from the propagated comments to facilitate various software engineering tasks such as code bug detection. Our contributions are as follows:

- We construct a comprehensive comment taxonomy from different perspectives with various granularity levels, and train six classifiers using three algorithms to automatically categorize comments into appropriate perspectives and granularity levels.
- We propose a novel bidirectional method of leveraging program analysis to propagate comments and leveraging comment analysis to facilitate bug detection, which achieves a seamless synergy of comment analysis and program analysis.
- We develop a prototype CPC based on the proposed idea, and evaluate it on 5 large real-world projects. The evaluation results demonstrate that 41573 new comments can be derived by propagation from other code locations with 88% accuracy. Among them, we can derive precise functional comments for 87 native methods that have neither existing comments nor source code. Our user study shows propagated comments are as useful as existing comments in helping developers.

### 2 MOTIVATION

Modern software provide abundant natural language (NL) comments and there is substantial existing work on analyzing NL comments and leveraging them in a wide range of software engineering applications. However, as far as we know, existing work hardly leverages program analysis techniques to derive, refine, and propagate comments systematically. Such propagated comments contain wealthy semantics beyond traditional artifacts that have been widely used in program analysis like data types. For example, by using program analysis techniques, we can pass comments on to code entities that are not commented and leverage information contained in the propagated comments to detect code bugs. Our overall goal is to achieve code-comment co-analysis: (1) program analysis propagates and updates code comments, and (2) code comments provide additional semantic hints to enrich program analysis.

Software developers tend to comment on different aspects of different code elements [58]. Comments of different perspectives entail different propagation rules. Consider the two comments "throws IndexOutOfBoundsException if the index is out of range (index < 0 || index >= size())." and "Shifts any subsequent elements to the left (subtracts one from their indices)." of method remove(int index). The former can be propagated through data flow while the latter cannot. Suppose we have an assignment o = list.remove(i), by propagating the first comment from the method definition (to the statement), we can know that if the condition \( i < 0 \) or \( i >= \text{size}() \) holds, the assignment statement will throw an IndexOutOfBoundsException, which can be used to check the code. However, the second comment describes the implementation details involving the data structure used in remove(int index), which would be misleading and make no sense if propagated to the assignment. Hence, the first step towards comments propagation and inference is to build a complete taxonomy for comments.

There have been efforts in software documentation classification [46, 58, 60, 61, 74]. The researchers in [58] propose a taxonomy based on the following four dimensions: comment contents, comment authors/users, comment locations (e.g., before a loop or in a macro), and comment composition time. In [46], researchers manually classify API documentation based on the knowledge types (e.g.,...
accessed later in the program (line 4), the same property should also hold. Since variable `all` has the property of allowing null elements, each of its elements `item` is permitted to be `null` (step ③). As the `size()` method of element `item` is invoked to measure the size, with `item` being `null`, it will cause null pointer access and hence trigger a `NullPointerException`. This is a new bug detected by our technique (step ④). In total, we detect 29 such bugs which cannot be detected by existing techniques since they only use information contained in existing comments which rarely comment on local variables especially control variables only used during iteration. All the 29 bugs have been confirmed and fixed by developers. In addition, we detect another 8 bugs based on our propagated comments.

**Comment Defect Detection.** Comments are critical for code understanding. They also serve as instructions/manuals for (third party) developers to utilize classes and methods. Defective comments can mislead developers and even incur critical bugs. Fig. 2 demonstrates a real-world case where comments are missing for native methods. Method `newArray()` is implemented using native code (line 08), and it has two arguments `componentType` and `length`. Although comments are highly desirable here due to the black-box nature of native implementation, there is no comment for the native method, which can potentially lead to bugs (e.g., pass `-1` to parameter `length`). Such native methods are implemented in other languages (e.g., C++ and assembly) where the source code, in general, is unavailable. Comments of these methods cannot be generated using existing techniques since they either summarize source code to infer comments, or analyze existing software repositories and use the comments from similar code. We showcase how our technique can address this problem using the example in Fig. 2. Firstly, there is only one statement in method `newInstance()` (line 03). Hence, we can apply rule `PROPERTY-METHOD2ST` (detailed in Section 6.1) to propagate the `what-comment` (lines 01-02, meaning the functionality associated with method `newInstance()`) to the statement at line 06 (step ①). Secondly, the statement at line 06 only invokes the native method `newArray()`, which satisfies the preconditions of rule `PROPERTY-ST2CALLER` (detailed in Section 6.2). Thus, the comment can be further propagated from the statement at line 06 to its callee method `newArray()` (step ②). Through the propagation, a new comment can be generated for the native method, specifying the functionality of `newArray()` is to "Create a new array with the specified component type and length.". Using our technique, we are able to infer comments for 87 native methods that have neither source code nor comments in JDK. Note that these native methods may be invoked by many other Java methods such that their generated comments can be used to help these invocations.

**Wrong Propagation Without Classification.** A comment taxonomy is vital for comment propagation as different kinds of comments convey different semantic perspectives. As such, some of them cannot be directly propagated. For instance, even if two code snippets are exactly the same, propagating comment from one to the other may be problematic. Consider the two code snippets in Fig. 3(a) and Fig. 3(c). The method in Fig. 3(c) has a property comment "This method will block until the byte can be written.". Although the two code snippets are syntactically identical, we cannot propagate the property comment from the method in (c) to the method in (a). This is because the method invoked at line 4 of (a) and that invoked at line 8 of (c) have different implementations.
which have different characteristics. Specifically, line 4 in (a) calls the write method in Fig. 3(b) that is non-blocking. In contrast, line 8 in (c) calls the write in Fig. 3(d) that is blocking, indicated by the "synchronized" keyword in Fig. 3(e). Therefore, it is incorrect to propagate the aforementioned property comment. However, existing techniques [84, 85] use the comment in (c) as comment for the method in (a), as they work by identifying code clones and sharing comments across all clones. In contrast, our technique does not allow propagating property comments in such cases.

3 THE TAXONOMY OF COMMENTS

As aforementioned, taxonomy is critical for comment propagation. However, existing taxonomies cannot be leveraged to facilitate comment propagation due to two main reasons. The first one is that comments are not associated with the corresponding code entities, making it impossible to leverage program analysis to propagate comments. For example, we need to make sure a comment is commenting on a variable before we can propagate it through a definition-use relation of the variable. The second reason is that the taxonomies are not designed to be coupled with program analysis and comments in a category (by existing work) tend to describe multiple perspectives of a code entity such that it is unclear how to propagate such comments.

Hence we propose to construct a comment taxonomy by classifying comment texts based on two dimensions: code entity and content perspective, where code entity means elements like classes and methods and content perspective means functionalities, rationales, implementation details, etc. Such a taxonomy is vital since different comments describe different code entities from different perspectives (e.g., what, why, and how) which entail different propagation rules. To develop a comprehensive and rigorous taxonomy, we performed a content analysis which is a methodology for studying the contents of documents and communication artifacts [55] (Section 4). Our final taxonomy is illustrated in Table 1. The first column lists the code entities, namely, class, method, statement, and variable which are the subjects that are commonly commented by developers. For each code entity, we are interested in the following five perspectives: what, why, how-it-is-done, property and how-to-use.

What. The what perspective provides a definition or a summary of functionality of the subject and/or its interface. Critical semantics can be extracted from what information, such as security sensitivity, which is important for vulnerability identification. By reading such type of comments, developers can easily understand the main functionality of the corresponding code entity, without diving into (implementation) details. For example, the comment "Pushes an item onto the top of this stack" in the seventh row of Table 1 describes the main functionality of method push(E item).

Why. The why perspective explains the reason why the subject is provided or the design rationale of the subject. There are two scenarios in which why perspective is important. First, it helps developers understand methods whose objective is masked by complex implementation. For example, the comment "Helps prevent entries that end up in the same segment from also ending up in the same bucket" of the method segmentShift() conveys why we need this method, while from the implementation we can only tell it moves some objects. Second, there exist multiple methods that look similar but serve different purposes. In this case, developers often provide why comments to point out why these similar methods are needed and explain why they are not plain redundancy.

How-it-is-done. The how-it-is-done perspective describes the implementation details like the design or the work-flow of the subject. Such information is critical for developers to understand the subject, especially when the complexity is high. Detecting inconsistencies between how-it-is-done comments and implementation is a way to find bugs. Moreover, many program analyses avoid analyzing complex library implementation due to the entailed space and time overhead. Instead, program analysis developers often rely on how-it-is-done comments to synthesize (much simpler) code snippet to model library functionalities. For example, the comment "Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices)." of method add(int, E) implies that we can implement the functionality by manipulating an array within a loop statement (e.g., "for (int i=size-1; i>index; i--) elements[i]=elements[i-1]"). While the original library code is usually highly optimized (and hence complex and difficult to analyze), the model code is simple and much easier to analyze.

Property. The property perspective asserts properties of the subject, e.g., pre-conditions/post-conditions of a function and even some statements. Pre-conditions specify the conditions that should hold in order to use the subject while post-conditions indicate the result of using the subject. Such comments are of importance as they can be used in many software engineering tasks, such as program verification, defect detection and program testing. For example, the comment "The index must be a value greater than or equal to 0." of the variable index (a parameter of setElementsAt(E, int)) specifies a pre-condition index ≥ 0 that must be satisfied for method setElementsAt(E, int) to work properly.

How-to-use. The how-to-use perspective describes the expected set-up of using the subject, such as platforms and compatible versions. For example, the comment "But using this class, one must
We collected a sample set of natural language comments from four classes and from 43 to 310 KLOC, and 30% of the lines of code open sourced. The size of the projects varies from 450 to 2500.

4.2 Coding Procedure

In this section, we illustrate the coding procedure [51] that we followed to construct the sample set. In total, we collected 5000 comment units. For each source file, we randomly sampled comments from each kind of code element in proportion to its subject (the type of code entity) and analyze its content (e.g., to identify information like the functionality). Each comment unit may target at different code entities and fall into multiple content categories. For example, the comment "Returns the head of this deque, or null if this deque is empty" of method pollFirst() not only describes the functionality (what) of this method, but also implies the implementation (how-it-is-done) of this method. In such cases, the coders would mark the comment with two labels. As mentioned earlier, it is possible that the coders would identify some categories that were not in the initial taxonomy. Thus the to-be-completed taxonomy was shared among coders via an online spreadsheet, which allowed each coder to add new categories to the taxonomy. Once a new category was identified and included in the taxonomy, all the other coders would be notified and they would discuss and verify. If all the coders agree on the new category, then the taxonomy would be updated to include the new one.

As we manually processed comments, it is inevitable to introduce subjectivity. To minimize such subjectivity, we utilized cross-verification by assigning each comment unit to two different coders. When disagreement occurs, all the coders would involve to have an open discussion to resolve it. Since to what extend two coders agreed on the categories of each comment unit is a direct measurement of both the reliability of the comment taxonomy and setting of the procedure [57] and made use of four coders (participants in a coding procedure). All the coders had at least four years of programming experience and were acquainted with program comments. With the intention of specifying a starting coding framework, one coder carried out a pilot study on 200 comments of the sampling set by identifying different content perspectives with the corresponding characteristics. This study brought forth the initial comment taxonomy which covered the majority of the final taxonomy. Some categories shown in Table 1 did not occur in the 200 comments and we refined the taxonomy in the later phase. Based on the initial taxonomy, this coder held a 60-minute session to train the remaining coders either on-site or through video conferences. During the session we discussed the meaning and the examples of each category and clarified misunderstandings that arose.

The 5000 comment units were randomly and evenly assigned to all the coders, which ensured that each coder categorized comments of all the four projects. For each comment unit, the coders identify its subject (the type of code entity) and analyze its content (e.g., to identify information like the functionality). Each comment unit may target at different code entities and fall into multiple content categories. For example, the comment "Returns the head of this deque, or null if this deque is empty" of method pollFirst() not only describes the functionality (what) of this method, but also implies the implementation (how-it-is-done) of this method. In such cases, the coders would mark the comment with two labels. As mentioned earlier, it is possible that the coders would identify some categories that were not in the initial taxonomy. Thus the to-be-completed taxonomy was shared among coders via an online spreadsheet, which allowed each coder to add new categories to the taxonomy. Once a new category was identified and included in the taxonomy, all the other coders would be notified and they would discuss and verify. If all the coders agree on the new category, then the taxonomy would be updated to include the new one.

As we manually processed comments, it is inevitable to introduce subjectivity. To minimize such subjectivity, we utilized cross-verification by assigning each comment unit to two different coders. When disagreement occurs, all the coders would involve to have an open discussion to resolve it. Since to what extend two coders agreed on the categories of each comment unit is a direct measurement of both the reliability of the comment taxonomy and
The trained word embedding will be used to train a word classifier introduced in Section 5.3. The last column describes each feature. Note that all the features are automatically extracted, meaning that no human effort is required to use our trained classifiers to categorize comments.

Feature `tokenNum` is the number of tokens contained in a cleaned comment, and `classNum` is the number of classes mentioned in a comment. A comment which mentions more classes tends to have a higher probability to be an explanation of implementation details, indicating itself to be a *how-it-is-done-comment*. The remaining six features are extracted from parse trees and Stanford dependencies generated by Stanford Parser [40, 65]. The Stanford parser parses a sentence and determines Part-Of-Speech (POS) tags [41] associated with different words and phrases. Parse trees represent grammatical structure of sentences and Stanford dependencies represent grammatical relations between words in a sentence. For example, Fig. 4(a) shows the Stanford parse tree of the comment "Gets the value for the Entry" where NP, VP, PP, etc., are POS tags. Feature `tree` is a string representation of a parse tree which is composed of nodes that are traversed using breadth-first search (BFS). For example, Fig. 4(b) is the BFS of the tree in Fig. 4(a). The features `NPNum`, `VPNum` and `PPNum` count the number of noun phrase (NP) nodes, verb phrase (VP) nodes and prepositional phrase (PP) nodes, respectively. Stanford parser also provides dependency types for each pair of adjacent words [22]. We extract the relations `caseNum` and `preconjNum` which are the number of type `case` and type `preconj` contained in a sentence. The `case` relation is used for any case-marking element which is treated as a separate syntactic word (including prepositions, postpositions, and clitic case markers) [2]. `Preconjuncion` is the relation between the head of an NP and a word that appears at the beginning bracketing a conjunction and puts emphasis on it [10], such as "either", "both", "neither". The six features are utilized since based on our experiments, we observe that these six features have positive importance on the classification while the other elements contained in parse trees and dependencies have little positive importance or even have negative importance.

### 5.2 Feature Extraction

To train models to classify comments, we extract eight features shown in Table 2. The first column lists the features, and the second column gives the type of each feature, namely numeric and string. The last column describes each feature. Note that all the features are automatically extracted, meaning that no human effort is required to use our trained classifiers to categorize comments.

Feature `tokenNum` is the number of tokens contained in a cleaned comment, and `classNum` is the number of classes mentioned in a comment. A comment which mentions more classes tends to have a higher probability to be an explanation of implementation details, indicating itself to be a *how-it-is-done-comment*. The remaining six features are extracted from parse trees and Stanford dependencies generated by Stanford Parser [40, 65]. The Stanford parser parses a sentence and determines Part-Of-Speech (POS) tags [41] associated with different words and phrases. Parse trees represent grammatical structure of sentences and Stanford dependencies represent grammatical relations between words in a sentence. For example, Fig. 4(a) shows the Stanford parse tree of the comment "Gets the value for the Entry" where NP, VP, PP, etc., are POS tags. Feature `tree` is a string representation of a parse tree which is composed of nodes that are traversed using breadth-first search (BFS). For example, Fig. 4(b) is the BFS of the tree in Fig. 4(a). The features `NPNum`, `VPNum` and `PPNum` count the number of noun phrase (NP) nodes, verb phrase (VP) nodes and prepositional phrase (PP) nodes, respectively. Stanford parser also provides dependency types for each pair of adjacent words [22]. We extract the relations `caseNum` and `preconjNum` which are the number of type `case` and type `preconj` contained in a sentence. The `case` relation is used for any case-marking element which is treated as a separate syntactic word (including prepositions, postpositions, and clitic case markers) [2]. `Preconjuncion` is the relation between the head of an NP and a word that appears at the beginning bracketing a conjunction and puts emphasis on it [10], such as "either", "both", "neither". The six features are utilized since based on our experiments, we observe that these six features have positive importance on the classification while the other elements contained in parse trees and dependencies have little positive importance or even have negative importance.

### 5.3 Algorithms and Evaluation

To train classifiers to categorize comments into different code entities and different perspectives, we leverage the following three algorithms: decision tree [64], random forest [18] and convolutional neural network (CNN) [38]. The three algorithms are frequently used to train classification models and they are proven to have high accuracy in classification [39, 74]. The decision tree algorithm and the random forest utilize the extracted features to train models while the CNN algorithm does not use any feature. As mentioned earlier, a comment may fall into different categories and thus we train multi-label classification models [83] for both perspectives and code entities. The multi-label classification problem is transformed into a set of binary classifications and each binary classification checks whether a comment can be classified into one category.

To evaluate the classifiers, we apply the standard 5-fold cross validation [42], namely we randomly select 20% comments collected in Section 4 as the testing set and the remaining comments as the training set. The performance of perspective/code entity classification is summarized in Table 3. The columns `DTC`, `RFC` and `CNN`
respectively show the performance of decision tree, random forest and CNN. The four metrics we use are Precision, Recall, F1 Score and Hamming Loss which are calculated using sklearn metrics [11]. Specifically, Precision measures the ability of the classifier correctly labels a comment, and it is calculated as TP/(TP + FP) where TP is the total number of correctly classified comments and FP is the total number of comments that are classified into wrong categories. The metric Recall measures the ability of the models to correctly classify all the comments that belong to one category and it is calculated as TP/(TP + FN) where FN is the number of comments that are not classified as one category while in fact they belong to that category. The F1 Score is a weighted average of precision and recall and the Hamming Loss is the fraction of labels that are incorrectly predicted. For the first three metrics, the higher the better while for the hamming loss, the lower the better.

From this table, we can see that the three algorithms achieve high precision, recall and F1 score and low hamming loss in classifying comments into correct perspectives, which indicates the effectiveness of our classifiers. For the code entity classification, the decision tree and the random forest algorithms have high precision, recall and F1 score and low hamming loss. In contrast, CNN has relative lower precision, recall and F1 score and relative higher hamming loss compared with the other two algorithms. The classification of code entity is more sensitive to input features. CNNs performs their own feature abstraction, which may miss important features.

6 PROPAGATION

In this section, we will introduce the rules that are used to propagate comments based on their corresponding code entities and perspectives. These propagation rules achieve the goal of leveraging program analysis techniques to update existing comments, infer new comments and associate comments with code. We have different rules for different code entities and perspectives. Each rule is headed by its name, followed by a fraction with theominator the conditions and the denominator the derived comment.

6.1 Property-comment Propagation

The property-comment propagation rules are summarized in Fig. 5(a), which involve rules for propagating comments of different levels of granularity, namely class-, method-, statement- and variable-level.

Class-level Propagation. Rule Property-Instant defines the propagation between a class and its instantiation. That is, if a comment c is associated with a class o and a variable v instantiates class o, then the property-comment is propagated from class o to variable v. The expression c[v/o] means the occurrence of o in c is replaced with v. For example, the property-comment “permits all element, including null” of the class ArrayList is propagated to the instance all declared at line 1 in Fig. 1. Rule Property-Inher specifies that if (1) a comment c is associated with superclass q, and (2) there is an inheritance relation between subclass p and superclass q, then property-comment c is propagated from superclass q to subclass p with the class name q (superclass) occurring in c substituted with the class name of p (subclass). Rule Property-Impl is analogous to rule Property-Inher, where the property-comment c associated with interface i is propagated to its implementation class o.

Method-level Propagation. Rule Property-Callee2St is applied to propagate a property-comment if a comment c contains properties regarding a callee method m and p is invoked by a statement s, then the property-comment c is propagated from the callee method m to the statement s with the formal parameters p fn in c substituted with the actual parameters ap used in s. Rule Property-Method2St defines the scenario that a comment c associated with a method m is propagated to a statement s when the statement is the only statement in m. For example, in Fig. 2, the statement return newArray(componentType, length) at line 6 is the only statement of the method newArray() and thus we can propagate the property-comment “@exception NegativeArraySizeException if the specified code length is negative.” at lines 3-4 to the statement at line 6.

Statement-level Propagation. Rule Property-St2Callee specifies that if (1) a statement s invokes a callee method m, and (2) s has no additional operations other than returning the result of the callee m, then the property-comment c is propagated from the statement s to the callee method m with the actual parameters ap substituted with the formal parameters p. Consider the aforementioned statement (line 6 in Fig. 2) which invokes the method newArray() and does not have operations except returning the result of newArray(). Since the two conditions are met, we can infer a new property-comment for the native method newArray() by propagating “@exception NegativeArraySizeException if the specified code length is negative.” from the statement to the callee newArray(). We can observe that comments of a callee method can be propagated from a callee method via the invocation statement based on Rule Property-Method2St and Rule Property-St2Callee.

Rule Property-St2Method defines propagating comments from a statement s to a method m which contains s, under the condition that the set of actual variables ap contained in s is a subset of the parameters of m. Suppose that the property-comment c describes a property of a variable that is not a parameter of m and thus it would be inappropriate for c to be a comment of m.

Variable-level Propagation. Variable-level rules include two cases: definition-use and container-element. Rule Property-DefUse defines the case that if a comment c is associated with a variable v and v is used in code u, then the property-comment c is propagated from definition v to use u. Rule Property-Container specifies that if (1) a comment c is associated with container l and l has element e, then a element-related property-comment is propagated from container l to each element e.

6.2 What-comment Propagation

The rules to propagate what-comment are shown in Fig. 5(b). Similar to the rules of property-comment, they are also categorized based on classes, methods, statements and variables.

Class-level Propagation. Rule What-Inher is similar to Property-Inher, where a what-comment c associated with a superclass q is propagated to a subclass p. Rule What-Impl is also similar.
Method-level Propagation. Rule \textit{What-Caller2St} specifies that if (1) a comment \(c\) is associated with a method \(m\), and (2) there is a method invocation relation between a statement \(s\) and the callee method \(m\), then the \(\textit{what-comment}\) is propagated from the callee \(m\) to the statement \(s\) with the formal parameters \(fp\) in \(c\) substituted with the actual parameters \(ap\). Rule \textit{What-Method2St} denotes the propagation from a method \(m\) to a statement \(s\) inside. Two preconditions are required to be satisfied for the propagation. The first condition is that the statement \(s\) is the last statement of method \(m\). The second one can be either 1) there are no statements before \(s\), namely \(\text{preSts} = \emptyset\) or 2) all the previous statements are for exception handling (\(\text{preSts} \subseteq \text{exSts}\)). If the two conditions are met, the \(\textit{what-comment}\) can be propagated from the method \(m\) to the statement \(s\) Consider the example shown in Fig. 2. The statement at line 6 is the last statement of the method \textit{newInstance()} (lines 5-7), meaning the first condition is satisfied. Also the method body does not have statements before line 6, meaning the second condition holds. Hence the \(\textit{what-comment}\) "Creates a new array with the specified component type and length," (propagated to the statement based on Rule \textit{What-Method2St}) to \textit{newArray()} at line 8. Rule \textit{What-St2Method} is symmetric to rule \textit{What-Method2St} and discussion is elided.

Variable-level Propagation. Variable-level propagation is defined by rule \textit{What-DefUse}. That is, if variable \(v\) is associated with a comment \(c\) and there is a definition-use relation between \(v\) and \(u\), then \(c\) is propagated from definition \(v\) to use \(u\).

6.3 How-it-is-done-comment Propagation

The propagation rules for \textit{How-it-is-done-comment} are given in Fig. 5(c) and they only involve method-level propagation. Comments can be propagated in other levels, but in practice, most \textit{How-it-is-done-comments} are in method-level. The first rule \textit{How-Clone} specifies the scenario that if (1) a \textit{how-it-is-done-comment} \(c\) is associated with a method \(m\), and (2) the method body of \(m\) is the same as the body of another method \(m'\), then \(c\) is propagated from \(m\) to \(m'\). The second rule \textit{How-DefType} specifies that if method \(m'\) is a code clone of method \(m\) but with different types of variables or formal parameters, then comment \(c\) is propagated from \(m\) to \(m'\) with the type information substituted.

7 EVALUATION

We implement a prototype CPC, leveraging the Eclipse JDT toolkit [6] and the code clone tool Nicad [68], and empirically evaluate it to address the following questions:

\textbf{RQ1}: How effective is CPC in propagating comments of different perspectives and code entities?

\textbf{RQ2}: How useful is CPC in helping developers?
Table 4: Comment Propagation Summary

<table>
<thead>
<tr>
<th>Perspective</th>
<th>Project</th>
<th>#c</th>
<th>#m</th>
<th>#ec</th>
<th>#pc</th>
<th>#cmnt</th>
<th>#print</th>
<th>#cmnt/ #print</th>
<th>#print</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>JDK</td>
<td>998</td>
<td>17727</td>
<td>21147</td>
<td>39274</td>
<td>9333</td>
<td>75.11%</td>
<td>2191</td>
<td>18.02%</td>
</tr>
<tr>
<td></td>
<td>Joda-Time</td>
<td>247</td>
<td>2677</td>
<td>3151</td>
<td>4222</td>
<td>3901</td>
<td>73.39%</td>
<td>372</td>
<td>20.96%</td>
</tr>
<tr>
<td></td>
<td>ApacheDB</td>
<td>518</td>
<td>6140</td>
<td>1940</td>
<td>8425</td>
<td>2718</td>
<td>88.28%</td>
<td>259</td>
<td>8.41%</td>
</tr>
<tr>
<td>What</td>
<td>JDK</td>
<td>219</td>
<td>5011</td>
<td>2344</td>
<td>4393</td>
<td>261</td>
<td>974</td>
<td>1392</td>
<td>16285</td>
</tr>
<tr>
<td></td>
<td>Joda-Time</td>
<td>193</td>
<td>3508</td>
<td>1098</td>
<td>2552</td>
<td>779</td>
<td>82.43%</td>
<td>57</td>
<td>6.03%</td>
</tr>
<tr>
<td></td>
<td>ApacheDB</td>
<td>768</td>
<td>10841</td>
<td>12927</td>
<td>330</td>
<td>133</td>
<td>0.83%</td>
<td>58</td>
<td>45.67%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>83</td>
<td>45</td>
<td>133</td>
<td>0.83%</td>
<td>133</td>
<td>0.83%</td>
<td>58</td>
<td>45.67%</td>
</tr>
</tbody>
</table>

Table 5: Comment Propagation Accuracy

<table>
<thead>
<tr>
<th>Perspective</th>
<th>#cmnt</th>
<th>#print</th>
<th>#cmnt/#print</th>
<th>#print</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>100.00%</td>
<td>76.00%</td>
<td>83.00%</td>
<td></td>
</tr>
<tr>
<td>What</td>
<td>100.00%</td>
<td>71.00%</td>
<td>70.00%</td>
<td></td>
</tr>
<tr>
<td>How-it-is-done</td>
<td>100.00%</td>
<td>75.00%</td>
<td>70.00%</td>
<td></td>
</tr>
</tbody>
</table>

RQ3: How effective is CPC in improving comments?

RQ4: How effective is CPC in detecting code bugs?

The evaluation was conducted on a machine with Intel(R) Core(TM) i7-8700K CPU (5.00GHz) and 32GB main memory. The operating system is macOS High Sierra 10.13.6, and the JDK version is 8.

7.1 Effectiveness in Comments Propagation

To answer RQ1, we propagate property-comments, what-comments and how-it-is-done-comments in five projects. The results are summarized in Table 4, which presents the comment perspective (column 1), the projects (column 2), the number of classes/methods whose comments are propagated (columns #c and #m), the number of existing comments/propagated comments (columns #ec and #pc), the similarity between an existing comment and an propagated comment (columns #cmnt and #print), the percentage of comments with distance larger than 0.5 in the remaining columns show the accuracies of different distance ranges, summarized in Table 5. The first column gives the perspective of comments and the remaining columns show the accuracies of different distance ranges. Due to the large number of propagated comments, we cannot manually check all of them. Instead, for each distance range, we randomly sampled 500 comments of each perspective and manually checked whether the propagated comments are correct or not. If a propagated comment is inconsistent with the source code, it is considered as false positive.

Table 5 shows that we achieve 100% accuracy when the distance is 0 and an average of 75% accuracy when the distance is larger than 0. This demonstrates that our propagation technique is effective in inferring comments. Note that even though the distance is larger than 0 or even larger than 0.5, it does not mean the propagated comments are incorrect since the same semantics can be expressed using different sentences. For example, the comment “Returns the node; or null if not found” is propagated to method remove() of class ConcurrentSkipListMap and this method has an existing comment “Returns the previous value associated with the specified key; or null if there was no mapping for the key.” The two comments are literally quite different, but they have the same semantics.

7.2 Usefulness in Helping Developers

To answer RQ2, we conducted a user study involving 14 users (6 graduate students and 8 developers from industry) to participate. We randomly selected 80 code entities that have both existing comments and propagated comments (with a total of 160 comments). The generated comments are propagated from other places and must be syntactically different from the existing ones. They are mainly from Commons Collections, JDK, and Guava, and have even coverage for the three comment types. To diversify our selection, these code entities are selected from different source files. To avoid bias, we mix the propagated comments and the existing comments, and thus the users are unaware of whether a comment is propagated or existing. For each comment, we provide the corresponding code, and ask users to evaluate the comments from the following three perspectives: Meaningfulness (is a comment of high quality in helping developers understand code), Consistency (is a comment consistent with code), and Naturalness (does a comment effectively convey information as a natural language sentence).
Table 6: Comment Propagation Improvement

<table>
<thead>
<tr>
<th>Project</th>
<th>Property</th>
<th>What</th>
<th>How-it-is-done</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#N</td>
<td>#I</td>
<td>#W</td>
</tr>
<tr>
<td>JDK</td>
<td>26862</td>
<td>11</td>
<td>243</td>
</tr>
<tr>
<td></td>
<td>1589</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>242</td>
<td>n.a.</td>
<td>0</td>
</tr>
<tr>
<td>Collections</td>
<td>2404</td>
<td>11</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>93</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>34</td>
<td>n.a.</td>
<td>0</td>
</tr>
<tr>
<td>Guava</td>
<td>5344</td>
<td>2</td>
<td>447</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>n.a.</td>
<td>0</td>
</tr>
<tr>
<td>Joda-Time</td>
<td>2757</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>79</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>n.a.</td>
<td>0</td>
</tr>
<tr>
<td>ApacheDB</td>
<td>1607</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>n.a.</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 6: User Study Comparison Result

The users are asked to evaluate each comment based on the widely-adopted five-point Likert scale [45], and the scores 1, 2, 3, 4, 5 separately represent strongly disagree, disagree, neither agree nor disagree, agree and strongly agree. Note that the numerical results of these questions are not important as they are dependent on the quality of the original comments (recall our propagated comments also originate from existing comments). Instead, the comparative results of the two kinds of comments are important. Fig. 6 shows the comparison results between propagated comments (blue) and existing comments (red): 3.88 vs 3.86 for consistency, 3.85 vs 3.84 for meaningfulness, and 3.98 vs 4.00 for naturalness. Overall, the results indicate propagated comments align well with existing ones in terms of quality. Further inspection shows that the slightly worse results regarding naturalness are due to our sampling bias: the propagated comments are 16.7% shorter than the existing comments and the users seem to consider longer comments are more natural.

7.3 Effectiveness in Improving Comments

To answer RQ3, we evaluate the effectiveness of our comment propagation in three aspects: 1) inferring new comments for code entities that do not have existing comments; 2) identifying incomplete comments which may be misleading for users or developers; and 3) detecting wrong comments that might lead to bugs. The result is shown in Table 6, and the columns #N, #I and #W present the number of new comments, incomplete comments and wrong comments, respectively. Here a new comment means the code entity where the comment is propagated to does not have any comment before. Note that we do not give the number of incomplete how-it-is-done comments since it is unnecessary and impractical to comment all the implementation details.

Based on the number of new comments shown in Table 6 and the accuracy in Table 5, we can see that our technique can effectively generate new comments, which can be further used to facilitate understanding and maintain documentation. By manually checking some of the newly-generated comments, we find that many comments describe exceptional behaviors including the type of exception and the corresponding exception-trigger condition. Such comments are usually considered very important since a majority of bugs are caused by triggering exceptions. For example, we can generate the new comment "throws IllegalArgumentException if the size is less than 1" for constructor CircularFifoQueue(Collection), which conveys that the parameter collection should have a size larger than 0. Without such a comment, bugs are easily introduced since very few methods have such requirements and developers are insensitive to them. Moreover, among the new comments, precise functional comments are inferred for 87 native methods that have neither comments nor source code. These comments can serve as manuals for developers to leverage these native methods.

We also identify 11 incomplete comments which can be seen as inconsistencies between comments and code. Our propagated comments can be used to complement existing comments to address such inconsistencies and reduce the risk of introducing bugs when the code is used. For example, "Returns true if this list changed as a result of the call," is the existing comment of method addAll() in class RoleList, and "Returns true if the RoleList specified is null." is one propagated comment of the method. By analyzing the code, we can see the propagated comment is correct, meaning the existing one is incomplete. In addition, we detect many wrong comments that would be misleading and even lead to bugs. For example, in the project Apache Commons Collection, we generate the propagated comment "throws IndexOutOfBoundsException if index < 0 or index >= size()" which is inconsistent with the existing comment "throws IndexOutOfBoundsException if index < 0 or index > size()" of method setIterator in class CollatingIterator. We confirmed our propagated comment is correct, and developers also confirmed this and corrected the existing wrong comment [5].

7.4 Effectiveness in Bug Detection

To answer RQ4, we write a script to extract code whose propagated comments describe behaviors related to NullPointerException and IndexOutOfBoundsException based on buggy patterns. For NullPointerException, the buggy pattern is the code that does not check whether the return value of a method (whose comments state a null value may be returned) is null before dereferencing it. For IndexOutOfBoundsException, the buggy pattern is the code that does not check if the returned value of a method (whose comments state -1 may be returned) is -1 before using it to access an array.

Table 7 reports the bug detection results including, from left to right, the project, the project version, the number of detected bugs, the buggy method and whether the reported bug is confirmed. Due to the space limitations, not all the bugs are presented in the table. In total, our script reports 57 bugs. By manually checking them, we believe 37 of them are true bugs. We have reported the 37 bugs to the developers, among which, 30 bugs have already been confirmed and fixed by developers [4, 5], while the remaining ones await confirmation. For the false positives, the main reason is that our analysis script is not context-sensitive and hence cannot identify cases in which users will never pass parameters that trigger the function to return null or -1. It is a limitation of our scanner, not comment derivation.

8 THREATS TO VALIDITY

The threat to construct validity is the bias that may be introduced during the manual labeling of comments (Section 4). To mitigate this threat, each comment was categorized by two developers independently, and a third developer would manually resolve all cases
when two developers disagreed. We assessed the labeling reliability by measuring the inter-coder agreement (Section 4). In the future, we will further minimize the threat by inviting more developers to categorize comments. The threat to internal validity is the potential overfitting problem of the machine learning algorithms. To minimize this, we randomly selected 80% of the dataset as the training data and applied a five-fold cross validation. The threat to external validity is that it is plausible the classifiers produced on our training data would have low accuracy when categorizing comments from other projects. To alleviate this threat, we prepared labeled comments from four different software projects that cover different types of functionalities (e.g., Java collection framework and calendar system). While we believe that the comments from these software systems well represent comments in other software projects, we do not intend to draw any general conclusions. In the future, we will train the classifiers with more labeled comments of other kinds of systems to improve the generalizability. The user study was conducted with 14 users and 80 code entities. While we tasked each user with a lot of code and comment, we will extend the study to a larger user group.

9 RELATED WORK

Comment Classification. Researchers in [58] proposed a taxonomy based on meanings of comments and manually classified 1050 comments. They found 52.6% of these comments can be leveraged to improve software reliability and increase programmer productivity. In [52], researchers empirically studied API directives which are constraints about usages of APIs, and built a corresponding taxonomy. The authors of [46] leveraged grounded methods and analytical approaches to build a taxonomy of knowledge types in API reference documentation and manually classified 5574 randomly-sampled documentation units to assess the knowledge they contain. Based on this taxonomy, the researchers in [43] trained a classifier for each knowledge type and assigned only one label to each document unit based on nine features and their semantic and statistical combinations. In comparison, each classifier in our work classifies comments into different perspectives and code entities. The work [29] built a taxonomy of comments to investigate developers’ commenting habits while the work [74] studied comment categorization to provide better quantitative insights about comment quality assessment. Researchers in [60, 61] produced a taxonomy of comments and investigated how often each category occurs by manually classifying more than 2,000 code comments. Unlike them, we develop the taxonomy to treat a comment as an attribute of a code entity and thus we can leverage program analysis techniques to infer, propagate, update and reason about comments. It is unclear how to propagate comments based on existing taxonomies.

Comment Generation. There are efforts of generating comments from source code/code changes, based on manually crafted templates [13, 19, 47, 48, 53, 54, 66, 71–73], information retrieval [27, 28, 32, 84, 85], and machine translation [14, 31, 33, 35, 62]. The techniques [84, 85] are most closely related and they generated comments for a code snippet by using comments of its code clone. However, they did not distinguish between comments of different perspectives and thus may generate many wrong comments. Also they did not utilize techniques like data flow analysis to propagate comments, which is our novelty. Our technique differs from comment generation in a few aspects. Comment generation produces comments from code. However, different projects have different coding and comment styles. A generation technique trained on a set of projects or based on rules may not generate good comments on other projects. Instead of generating comments from code, we propagate existing comments to code entities that are not commented by leveraging program analysis. Our technique is less sensitive to such styles as it only classifies comments instead of generating them. Comment propagation is deterministic and rigorous through program analysis. Secondly, generating comments for complex code that even humans can hardly understand is error-prone. For such cases, our technique can leverage existing comments (from other places). Thirdly, evaluating quality of generated comments, such as their naturalness, is a hard challenge. Our technique is largely immune to this. Finally, our technique can propagate comments to methods without code while existing work requires code as input. On the other hand, comment generation and comment propagation are complementary. Through propagation, we can produce a much larger training set for generation techniques. Generated comments can be propagated through our technique.

Comment-Code Inconsistency Detection. Research has been conducted on improving API documentation maintenance such as reporting potential code-comment inconsistencies as code evolves [21, 67], detecting existing code-comment inconsistencies [77–79, 89, 91], and enriching documentation (e.g., with code samples) [26, 34, 37, 75, 76, 82]. They do not aim to explicitly propagate comments as first-class objects and thus our efforts are complementary.

10 CONCLUSION

We build a comprehensive comment taxonomy from different perspectives with various levels of granularity and propose using program analysis to propagate comments. We develop a prototype CPC. Our experiments show that CPC can generate 41573 new comments with 88% accuracy. The derived comments are used to detect 37 new code bugs in 5 real-world projects with 30 confirmed and fixed by developers. We also identify 304 defects in existing comments, including 12 incomplete comments and 292 wrong comments. Our user study confirms propagated comments align well with existing comments regarding quality.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This research was supported, in part by NSF-China 61802166, 61972193 and 61832009, DARPA FABA650-15-C-7562, NSF 1748764, 1901242 and 1910300, ONR N000141410468 and N000141712947, and Sandia National Lab under award 1701351. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.
REFERENCES


