ON-THE-FLY DISCOVERY OF LOCAL BUGS USING INCONSISTENCY ANALYSIS

Srinivasan H Sengamedu
AWS AI
sengamed@amazon.com

Qiang Zhou
AWS AI
zhouqia@amazon.com

Hangqi Zhao*
AWS AI
hangqizhao@gmail.com

ABSTRACT

Traditional bug detection mechanisms have focused on a limited set of important issues and have specialized detectors for each of them. As the code corpora continue to grow in size and complexity, newer opportunities for a developer to make mistakes emerge, leading to *long tail of local bugs*. Hence, we must investigate generalizable approaches that can detect such bugs. In this paper, we formulate and use the inconsistency principle that can be applied to discover bugs at arbitrary code granularity, for example at the package level. We experiment with two types of formulations: Pointwise Mutual Information (PMI) based and Sequence based approaches that respectively model smaller and larger contexts. The techniques learn code usage patterns from the code under analysis and apply the learnings on the same code – thereby enabling on-the-fly bug detection. Experiments are conducted with two different program representations: token-based and graph-based. We show how the different variations capture diverse and complementary types of issues. The technique is integrated with Amazon CodeGuru Reviewer. The approach has detected 12 types of bugs with 70% acceptance by developers in real-world code reviews.

1 INTRODUCTION

Software developers spend about half of their work time on debugging Britton et al. (2013), and therefore automated bug detection in software is of crucial importance for improving the efficiency of software development cycle. Different tools and methodologies have been developed to improve such efficiency and software quality. Most of the currently available tools are based on static analysis, and the primary disadvantage of these tools is the high number of false positives. Recently there have been several attempts at using machine learning based approaches for detecting bugs Pradel & Sen (2018), Islam et al. (2019). Such tools usually aim to detect *predefined* types of code bugs, or require a prior knowledge to define system rules, which typically involves mining and extracting appropriate rules from large code bases. However, the main issues of the machine learning based approaches are as follows.

1. Some coding conventions are code-based specific. For example, checking for a pre-condition or using a certain log-level for some types of messages are code-base specific. Violation of these conventions can have serious consequences and it is not possible to create generic models for detecting such bugs. In other words, there is a long tail of local bugs and many bug types are likely to be left out by generic detectors.
2. There are no large-scale labeled datasets for bug detection, and, as a result, supervised ML techniques are not readily applicable and we need to explore unsupervised approaches.
3. Since the bugs are local, it is essential to learn from code under analysis *during analysis* and apply the learnings *on-the-fly* in the same session (say, code review or repository scan).

*Currently with Twitter

4. Detecting specialized or new types of bugs requires effective communication of the rationale for the detected bugs to the developers. This is essential to the acceptance of the tool by the developers.

In this paper, we define and develop an unsupervised approach to bug detection. We call the underlying principle *inconsistency detection*. Software is repetitive and it is possible to *mine* specifications from analysis of code bases. While the mined specifications are often correct, violations of the specifications are not often bugs. For example, one precondition for string object is `str.length()>0` (see Nguyen et al. (2014)). While this is a valid precondition, performing this check before every use of the string is considered too *defensive* by developers. The reason is that the code uses are contextual and contextual analysis is required for detecting genuine violations. Hence *a naive enforcement of learned specifications leads to lots of false positives.*

Inconsistency principle is based on the following observation: while deviations from frequent behavior (or specifications) are often permissible *across* repositories, deviations from frequent behavior *inside a given repository* are worthy of investigation by developers and these are often are bugs or code smells. The following is an example of inconsistent behavior that is worthy of inspection: assume that access to the object *foo* is synchronized in 10 occurrences in a package and is not synchronized in one occurrence. As another example, if a package uses `str.length()>0` as a pre-condition in several cases and the condition is missed once or twice, in the best case, this is a bug and in the worst case, this is at least worthy of examination. Hence, our approach aims to automatically detect inconsistent code pieces that are abnormal and deviant from common code patterns in the **same** package, assuming that the common behaviors are correct. This anomaly detection approach does not need a prior knowledge and minimizes the human effort to pre-define sets of rules, as it can automatically and implicitly "infer" rules, i.e. common code patterns, from the source code.

While it is possible to detect inconsistencies in specific aspects (such as synchronization, exceptions, logging, etc.) of programs, *our goal is to build a single framework to detect inconsistencies in multiple aspects of programs*. Our approach is not restricted to certain types of code bugs but covers diverse categories of bugs.

We organize our experiments to answer the following research questions. The main conclusions for the questions are also summarized alongside.

The primary contributions of the paper are summarized below.

1. Propose *Inconsistency Principle* as an unsupervised approach to identify bugs in code bases.
2. Propose two approaches for detecting inconsistencies in code.
   (a) PMI-based approach on bigrams of tokens or graph nodes. This technique models local context.
   (b) Sequence-based inconsistency detection based on association rule derivation. This approach models longer code context.
3. Demonstrate that the proposed approaches are able to uncover diverse set of bugs in real-world code reviews. The overall precision of the approach is higher than the that reported in recent research Bryksin et al. (2018).

The proposed approaches detect issues on-the-fly in the sense that the learning of code patterns and the detection of bugs happen on the same repository during the analysis of the repository. *The paper uses several ideas – such as repetitiveness in code, locality of code patterns, and associative rule mining – that are familiar to the research community. We combine the ideas in a novel manner and show that the resulting approaches are effective in real-world code reviews and detect a broad range of issues. The related work section highlights the novel aspects and the experimental results section contains developer feedback.*

We organize the paper as follows. Section 2 expands on the idea of inconsistency principle. Section 3 discusses the feature extraction that are common to both the PMI and sequence based approaches. In Section 4, we present the PMI based approach on code tokens and on code graphs. Section 5 is devoted to the sequence based framework for inconsistency detection. In Section 6, we present experimental results on running our detection tool on private (company internal) and open source packages. In Section 7, we review related work on inconsistency detection in source code. Section 8 makes concluding remarks.

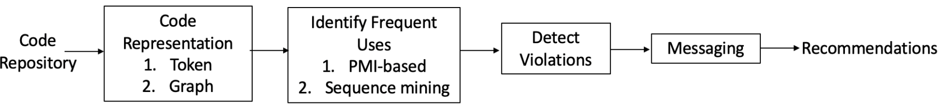

Figure 1: Inconsistency-based bug detection: Overview

2 INCONSISTENCY PRINCIPLE-BASED BUG DETECTION

An overview of inconsistency-based bug detection is shown in Figure 1.

Program Representation: We use two different representation of programs: tokens in program text as well as graph-representation of programs. The token-based representation requires no programming-language specific processing and hence scales to multiple programming languages. At the same time, the token-based representation does not capture proximities captured by data-flow and control-flow relations and hence misses several code usages. We have found that the inconsistencies detected in the two representations are complementary.

Identifying frequent uses: A key step towards detecting inconsistencies is identifying "consistencies" in code. There are many different forms of consistencies in code analysis, and in this paper we will focus on two particular types: bigram consistency and sequence consistency. For bigram consistency, the fundamental idea is that if two (or more) code elements <*a*> and <*b*> are strongly associated, i.e. they almost always appear together in the package, then any individual appearance of <*a*> (or <*b*>) without <*b*> (or <*a*>) could be a potential inconsistency as it is deviant from common code patterns. Namely, an implicit rule learned from this pattern is a => b: <*a*> must be followed by <*b*> in the code. A simple analogy in detecting misspellings in plain texts is that if a word a (e.g. *"Los"*) and a word b (e.g. *"Angeles"*) are strongly associated, then any scenarios *"Los"* is not followed by *"Angeles"* or vice versa are potential misspellings. In the context of source code, the code elements can be simply code tokens, or any nodes, links and derived features in the tree or graph representation of the source code. To find strong associations in code elements, we adopt a technique widely used in natural language processing called *Pointwise Mutual Information (PMI)* to identify collocations of code elements. To our knowledge, this is the first time PMI has been utilized for code bug detection.

For identifying co-occurring patterns of size larger than two, we use *itemset mining*. We use the frequent itemset mining framework followed by sequence alignment to identify the consistent patterns and detect outliers. The bigram and sequence consistency approaches are complimentary: one focuses on the very local pattern while the other one looks at much broader context. In terms of the actual bug findings there are almost no overlaps between these two.

Identifying violations: After obtaining strongly collocated code elements, we retrieve the source code snippets containing code elements that contradict such collocations through statistical analysis. These contradictions are outliers or counter examples that do not follow the common patterns. Such counter examples are not necessarily code bugs or code smells – they can be coincidences or different uses of code elements due to natural variations of code. To reduce such false positives, we have developed approaches including similarity analysis and program parsing to ensure that the inconsistent use of code tokens share similar or same context as the consistent ones.

Example based messaging: Detection of diverse types of issues requires appropriate communication of the issues to the developers. We pursue example based messaging and provide examples of frequent usages. Since the examples are from the same repository, developers relate very easily to the examples.

Messaging – Majority is not always right: The detector identifies frequent uses and violations of certain patterns. While violations are often bugs, in some cases the violations are correct usages and the *frequent usages are bugs*. One example scenario is rolling out the new version of an API. It is common that the updates are made only in a few places and not in all the uses. The inconsistency detector identifies the correct uses as outliers. Hence, in our messaging, we point out the inconsistency and ask the developer to resolve the inconsistency without claiming that the frequent usage is correct.

3 Program Representation

In general, the features used for inconsistency detection can be divided into two categories: tokens in the source code and paths in a graph representation of code.

3.1 Tokens

In natural language, a token is a string of contiguous characters between two spaces, or between a space and punctuation marks in the source code. It is natural in code that we see very long tokens as variable names or API calls, e.g., 'getCustomerAccountId'. We do not sub-tokenize long tokens as the scope of analysis is a package and the token-level vocabulary has manageable size at package-level.

3.2 Paths in MU Graphs

For a more elaborate analysis of the code, we utilize a graph representation of the code which includes data and control flow information of the source code. We use the graph representation called MU Graph Amann et al. (2019), which represents programs at statement and expression levels and captures both control and data flows between program elements. (Control flow represents the order of the execution and the data flow represents the flow of data along the computation.) We build MU Graphs at method level. (Figure 2 in Appendix A shows an example.)

The advantage of graph representation is that program elements (e.g., API calls) which are not adjacent in textual representation can be adjacent in the graph based on control- or data-dependence. Consider the following code snippet.

```
1 Account account = daoFactory.getAccountDao().getAccount(accountId);
2 Item item = daoFactory.getItemDao().getItem(itemId)
3 validate.notNull(account, item);
```

Though `getItem()` and `notNull()` are are not adjacent in the textual representation, they are adjacent in the MU Graph (Figure 2 in Appendix A) reflecting *semantic adjacency* while they. This relation helps us discover high-value inconsistencies.

4 PMI-based Inconsistency Detection

PMI (Pointwise Mutual Information) is widely used in Natural Language Processing (NLP) community to find collocations of *n-gram* in texts. We adopt this technique to extract strong associations between code tokens. The PMI for bigram (a, b) compares the frequency of the bigram to the frequency of the individual components of the bigram, and is defined as:

$$PMI(a,b) = \log \frac{p(a,b)}{p(a)p(b)} \tag{1}$$

where $p(a, b)$ is the joint probability of bigram (a, b) in the code corpus and $p(a)$, $p(b)$ are individual probabilities. Counting of occurrences and co-occurrences of tokens in the code corpus of a package is used to approximate the probabilities $p(a)$, $p(b)$ and $p(a, b)$. Higher value of PMI indicates stronger associations between tokens.

Detecting inconsistencies based on bigram tokens has the following three steps.

1. **Mine code patterns**. Mine from the package and extract all strongly associated bigram of code elements (*a*, *b*) through PMI analysis. This results in code patterns that *a* must be followed by *b*, or *b* must be preceded by *a*. The associated PMI value can be regarded as confidence score;

2. **Detect abnormal code**. Identify occurrences of code snippets in the same package that violate the code patterns. This results in a set of abnormal code that are candidate inconsistencies;

3. **Prune false positives**. Remove violations that are not true inconsistencies by comparing the context of violations and the context of common code patterns. The code elements in both contexts should share similar/same use cases and intent. This is done by analyzing a combination of code similarity and syntactic information obtained through code parsing.

Table 1: Path Features and the corresponding detectors

Path	Detector
All API calls within a method	Missing API
API calls associated with the same *receiver object*	Missing API-rcv
APIs and exceptions associated with the same `throw` edge	Try-Exception
Pair of APIs with producer and consumer relationship	Null-Check
Paths that connect input parameters to certain validation APIs	Missing Input Validation
Two neighboring API nodes with one node related to precondition	Missing Precondition Check
Two neighboring API nodes with one node related to logging API	Missing/inconsistent logging

Appendix B contains an illustrative example.

4.1 PMI on Graph Representation

We extend the PMI approach on token level to graph level by not directly utilizing the source code tokens but representing the program as a graph. Compared to token level approach, which can be regarded as a line-based detector, i.e. it compares all *lines* of code and detects inconsistent lines, the graph level approach looks at a bigger picture and more contexts. It detects inconsistencies in graph nodes, links or other features based on the structural information of graph representation of the source code. For example, considering each API call as a node in the graph representation, we can detect Missing API (MAPI) if a node is present in most cases of similar graph structures but is only missing in a certain graph. We have implemented such a MAPI detector based on this approach.

We represent program in the format of MU Graph. See Section 3.2. To detect MAPI, we extract all the nodes that represent an API call in the MU Graph of a Java method, and obtained a sequence of API calls within each method. Each item in the sequence is analogous to the token in a line of code in the token level approach. We can thus perform the same PMI analysis to the API call sequences within a package to extract all bigrams of API call that are strongly associated with each other. Those two APIs represent two nodes in the MU Graph that may be directly connected or share same paths. Similar to token level approach, the basic principle is that if two API calls A and B are strongly associated within the package, then A almost always should be called before (or after) B, otherwise it could be a potential missing call. A particular application of this could be detecting missing precondition check, as the precondition is almost always checked before invoking the method call it guards.

5 Detecting Inconsistency Based on Sequence Mining

With respect to Figure 1, sequence mining approach represents code at method-level by *paths in graph representation*. Itemset mining is used to identify frequent sub-paths and we detect violations of frequent sub-paths. For the purposes of specific messages in recommendations, we focus on specific types of paths. More specifically, sequence-based inconsistency mining has the following high-level steps.

Creation of itemsets: The itemsets are essentially paths in MU Graphs. While it is technically feasible to work with all paths, we found that developers relate to the recommendation better if the recommendation message is more explanatory. For the purposes of explanation, we work with *typed paths*: paths conforming to certain programming constructs. These constructs are shown in Table 1.

Itemset mining: We perform standard itemset mining on the collection of paths belonging to a given type.

Rule derivation: Once frequent itemsets are identified with support values, we construct rules satisfying minimum support and confidence values.

Sequence alignment: Since itemsets do not contain order information, the sequence alignment step incorporates order information. This step produces missing items and their locations.

Appendix C contains details of the steps.

Table 2: Categorized Token Level Detections – PMI-based approach

CATEGORY	USEFUL	NOT USEFUL	UNSURE	TOTAL	ACCURACY
MESSAGING	7	3	0	10	70.0%
LOGGING	7	2	1	10	70.0%
TYPO	9	1	0	10	90.0%
VISIBILITY	8	0	2	10	80.0%
TOTAL	31	6	3	40	77.5%

Table 3: Categorized Detections – Sequence Mining

CATEGORY	USEFUL	NOT USEFUL	UNSURE	TOTAL	ACCURACY
API	7	3	4	14	50.0%
API-RCV	17	4	5	26	65.0%
TRY-EXCEPTION	25	7	9	40	62.5%
NULL CHECK	24	11	5	40	60.0%

6 EXPERIMENTAL RESULTS

6.1 OFFLINE EVALUATION

PMI-based Approach
We evaluated our token level approach on internal Java packages. We have randomly selected 10 detections from 4 subcategories of code issues and conducted blind review by software developers on the correctness of the findings. 31 out of 40 findings are labeled as useful detections, 6 out of 40 are labeled as not useful, and the remaining 3 are unsure. The overall acceptance rate defined by number of useful detections over total number of detections is 77.5%. We summarize the performance of 40 detections in Table 2.

For the PMI on Graph representation, we have conducted human review by developers on the correctness of missing precondition detected by our tool. From 33 detections (obtained with a similarity threshold of 40) we got feedback from the developers: 14 of them are rated as Useful, 15 are rated as Not Useful and 4 are Unsure. The overall accuracy is 14/33 = 42.4%. As the precision of bug detection tools is often more important than recall because high false positive rate could hurt user experience, we increase the threshold. With a similarity threshold of 90, the precision reaches 74%.

Sequence-based Approach
We have performed offline validation of four detectors in this category including general missing API, missing API on the same receiver, null check, and try-exception, using sequence-based approach. They were tested on a mixture of open source GitHub and some internal packages. Table 3 summarizes the accuracies of these four detectors.

6.2 METRICS ON LIVE CODE REVIEW RECOMMENDATIONS

We have integrated this inconsistency detector family with our internal review tool and have validated with developers. The initial launch includes a *total of 12 detectors covering the following issues: typo, message, log level, declaration, pre-condition check, missing log, missing API (4 variants), null-check, and exception handing*.

Developers have an option of rating the code review comments with "USEFUL", "NOT USEFUL", and "NOT SURE". During a 4-week period, the recommendations from the detector got 44 feedback with 31 USEFUL, 12 NOT USEFUL, and 1 NOT SURE ratings. Hence the accuracy of recommendations is 70.5%.

An example recommendation is shown below.

For the code

```
LOG.warn("Document {} did not have a marketplace", id);
```

the recommendation was the following:
The detector found 3 occurrences of code with logging similar to the selected lines. However, the logging level in the selected lines is not consistent with the other occurrences. We recommend you check if this is intentional. The following are some examples in your code of the expected level of logging in similar occurrences:

```
LOG.info("Document {} does not have state", id);
```

```
LOG.info("Document {} did not have attributes", id);
```

One human code reviewer responded with “+1”, and the developer responded with “will fix. Didn’t realize as this was just copied from somewhere else.”

Another example is shown below.

For the code `document.deleteEntity(preEntity);`
the recommendation was the following:
We found 4 occurrences in your code of a similar method that calls Document.acceptChanges. Based on those occurrences, we expected it to be called after the selected line. We recommend you check if this is intentional. The following is an example in your code of when this method is called ...

The detection basically indicates a missing call of Document.acceptChanges after document.deleteEntity. One human code reviewer responded with "Looks like it is recommended by <an internal service> also in their CodingGuidelines to call document.acceptChanges() after deleting stale entities." This shows the detector is capable of detecting missing API calls in custom libraries as well as in popular ones.

Appendix D contains more examples from real-world code reviews.

6.3 Comparison with published results

As references for performance evaluation, Bryksin et al. Bryksin et al. (2018) used outlier detection techniques to detect anomalies in Kotlin code with 46/146 (31.5%) precision; Li et al. Li & Zhou (2005) developed PR-Miner to detect bugs in C code with frequency mining and achieved 52/127 (40.9%) precision if specifications are also counted as correct detections. NAR-Miner Bian et al. (2018) mines negative association rules. The reported true positive rate is 46.5% for rules and 4.8% for violations. Ahmadi et al. (2021) reports false positive rates exceeding 74% in several cases. See Table 4 in the paper. In contrast, our approach has a overall precision of about 70% in live code reviews.

Note that this is not an apples-to-apples comparison as the approaches have been tested on different datasets and on different languages. Despite the differences, this indicates that our approach achieved competitive performance compared to previous anomaly detection based techniques for code bug detection, and can be a good complement to previous tools for bug detection.

7 Related Work

Our work on inconsistency-based bug detection is at the intersection of several threads of research: repetitiveness in code, local structure of code, local structure of code changes, mining from code/code changes/execution traces, and finding anomalies.

Repetitiveness in code: Several classical papers have quantified the repetitiveness in code through entropy measurements, e.g., Hindle et al. (2012). While high entropy itself is correlated with bugs Ray et al. (2016), entropy has not been used for generating *actionable recommendations* to developers.

Mining from X: Associative rule mining and its variants have been used on several code-related artifacts: source code Acharya & Xie (2009); Zhong et al. (2009); Nguyen et al. (2014), code changes Meng et al. (2013); Nguyen et al. (2013), and execution traces Kumar et al. (2011); Yang et al. (2006). Li et al. (2006) identifies copy-paste errors at package-level.

Acharya & Xie (2009) addresses mining error handing specification for APIs and Zhong et al. (2009) mines API specifications for recommendations. Nguyen et al. (2014) addresses mining pre-condition

specifications of APIs. Unlike these approaches, the approaches proposed in this paper are capable of identifying package-level specifications of not just APIs but other aspects of code such as logging level, messages, etc.

Local structure of code and code changes: Code has local structure in terms of files and packages. In LASE Meng et al. (2013), the user provides examples of code changes and the edit script is inferred and applied on the package. Leveraging local structure provides additional gains. Language models leveraging local structure have shown increase in recommendation accuracy Franks et al. (2015). Ammonia Higo et al. (2020) mines project-specific edits.

Inconsistencies: Code inconsistencies have been investigated in previous studies. Engler et al. Engler et al. (2001) proposed a general approach to infer errors in system code that detects bugs as deviant behaviors. In Tomb & Flanagan (2012), Tomb et al. detected code inconsistencies via universal reachability analysis, which applies the weakest precondition operator to several modified versions of an input program. Schaf et al. Arlt & Schaf (2012) McCarthy et al. (2015) presented a tool called Joogie to detect infeasible code and a tool called Bixie to detect inconsistent code in Java program. They also proposed an algorithm Schaf et al. (2013) for explaining inconsistent code. Ocariza et al. Ocariza et al. (2017) implemented an anomaly detection based tool to detect inconsistencies in Javascript web application code. They found code patterns from tree-based data structures transformed from source code and detected violations of rules established from the code patterns. Similarly, Reiss Reiss (2007) found “unusual code” by reading a large corpus of code and building a library of common patterns from Abstract Syntax Trees. Dillig et al. Dillig et al. (2007) built a null dereference analysis of C programs based on semantic inconsistency inference. Monperrus et al. Monperrus et al. (2010) detected missing method calls in object-oriented software via identifying deviant code.

The papers closest to the research reported in this paper are Li & Zhou (2005), Bian et al. (2018) and Ahmadi et al. (2021). The first two use associative rule mining. Each bag consists of information at a function level. Unlike the path representation used by us, this information is too coarse. Hence the true positive rates reported are small and the techniques are applied on a small number (1-3) repositories. The last paper uses two-step clustering identify inconsistencies in C code. The technique works on LLVM bitcode (which requires compilation) and the detection latencies is of the order of hours. Our approach works on source code and is faster (most runs take less than 10 minutes). Section 6.3 shows that our approach has higher accuracy compared to these baselines.

Although we may work in the similar regime of anomaly detection, our work differs from previous studies and tools mainly in three aspects: 1) our framework works on *package* level instead of mining from a large corpus of packages. This not only improves the efficiency of detections as it does not require massive training, but also reduces false positives because patterns learned from various packages are prone to mistakes due to natural variations and diversity of code. Each package may maintain its own conventions. For instance, the use of an API call may be appropriate in one package but does not apply for another package. 2) We have proposed and developed a novel approach of finding common code patterns through PMI analysis or sequence mining to identify true deviant behaviors. 3) Unlike many previous tools that detect certain types of code issues, we developed a single model to cover a wider ranges of code bugs. Also, the results from our model could be further utilized for further fine tuning of specific classes of code issues.

8 Conclusions

In this paper we presented the idea of automatic bug detection using the inconsistency principle. We proposed two complimentary approaches based on package level inconsistency. The PMI-based method focuses on local patterns and sequence-based framework takes more contextual information into account. The two algorithms focus on different aspects of the problem and in general they could potentially provide complementary information to each other. We have developed 12 different bug detectors based on this inconsistency principle. They cover a wide variety of important and challenging bugs, including typo, null check, exception handling, logging, declaration, precondition check, and API usage, etc. This unsupervised approach does not require labeled dataset or massive training process. Meanwhile, it works on package level and does not require a prior knowledge on the codebase. We have demonstrated that our tool can detect diverse types of code issues and the performance is competitive to similar bug finding tools in previous studies. This family of detectors

have been launched internally and the recommendations have received 70% acceptance in addition to good qualitative feedback from the developers. The package level inconsistency is proven to be a powerful concept and there are many more bug detection scenarios that could be beneficial from this concept. Our approach could be easily extended to other programming languages such as Python, Go, as well as other representations of code such as Abstract Syntax Tree. In the future, we will continue to develop detectors based this inconsistency principle on more scenarios, and expand the feature extraction to more varieties of representation for code.

REFERENCES

Mithun Acharya and Tao Xie. Mining API Error-Handling Specifications from Source Code. In *FASE*, 2009.

Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases, VLDB*, 1994.

Mansour Ahmadi, Reza Mirzazade Farkhani, Ryan Williams, and Long Lu. Finding bugs using your own code: Detecting functionally-similar yet inconsistent code. In *30th USENIX Security Symposium*, 2021.

S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini. Investigating Next Steps in Static API-Misuse Detection. In *Mining Software Repositories*, 2019.

S. Arlt and M. Schaf. Joogie: Infeasible code detection for Java. In *CAV*, 2012.

Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. NAR-Miner: Discovering Negative Association Rules from Code for Bug Detection. In *FSE*, 2018.

Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. Reversible debugging software. *Cambridge Judge Business School*, 2013.

Timofey Bryksin, Victor Petukhov, Kirill Smirenko, and Nikita Povarov. Detecting anomalies in Kotlin code. In *ISSTA '18: Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, 2018.

Isil Dillig, Thomas Dillig, and Alex Aiken. Static error detection using semantic inconsistency inference. In *PLDI '07: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

Dawson Engler, David Y. Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SIGOPS Operating Systems Review*, 2001.

Christine Franks, Zhaopeng Tu, Premkumar Devanbu, and Vincent Hellendoorn‡. CACHECA: A Cache Language Model Based Code Suggestion Tool. In *ICSE*, 2015.

Yoshiki Higo, Shinpei Hayashi, Hideaki Hata, and Meiyappan Nagappan. Ammonia: an approach for deriving project-specific bug patterns. *Empirical Software Engineering*, 2020.

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *ICSE*, 2012.

Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. A comprehensive study on deep learning bug characteristics. In *arXiv:1906.01388*, 2019.

Sandeep Kumar, Siau-Cheng Khoo, Abhik Roychoudhury, and David Lo. Mining message sequence graphs. In *ICSE*, 2011.

Zhenmin Li and Yuanyuan Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, 2005.

Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. In *IEEE Trans. Software Engineering*, 2006.

Tim McCarthy, Philipp Rummer, and Martin Schaf. Bixie. Finding and understanding inconsistent code. In *ICSE*, 2015.

Na Meng, Miryung Kim, and Kathryn McKinley. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *Proceedings of 35th IEEE/ACM International Conference on Software Engineering*, 2013.

Martin Monperrus, Marcel Bruch, and Mira Mezini. Detecting missing method calls in object-oriented software. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, 2010.

Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, and Hridesh Rajan. A study of repetitiveness of code changes in software evolution. In *In Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, 2013.

Hoan Anh Nguyen, Robert Dyer, Tien N. Nguyen, and Hridesh Rajan. Mining preconditions of APIs in large-scale code corpus. In *FSE*, 2014.

Frolin S Ocariza, Karthik Pattabiraman, and Ali Mesbah. Detecting unknown inconsistencies in web applications. In *ASE*, 2017.

Michael Pradel and Koushik Sen. DeepBugs: A Learning Approach to Name-based Bug Detection. In *arXiv:1805.11683*, 2018.

Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the 'naturalness' of buggy code. In *ICSE*, 2016.

S. P. Reiss. Finding unusual code. In *Proceedings of the International Conference on Software Maintenance*, 2007.

M. Schaf, D. Schwartz-Narbonne, and T. Wies. Explaining inconsistent code. In *ESEC/FSE*, 2013.

A. Tomb and C. Flanagan. Detecting inconsistencies via universal reachability analysis. In *ISSTA*, 2012.

J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Terracotta: mining temporal API rules from imperfect traces. In *ICSE*, 2006.

Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and Recommending API Usage Patterns. In *ECOOP*, 2009.

Appendix

A Graph Representation

Figure 2 shows the an example code snippet and its graph representation. The graph representation is used with both PMI-based approach (Section 4.1) and sequence-mining based approach (Section 5).

B Illustrative Example for PMI-based detection

We will discuss each step in more detail in the next a few subsections. To illustrate this process, a simple token level example that we detected is shown below.

violation:

```
private static final String newStage = AppConfig.getDomain();
```

normal (10 times, shown 1):

```
private static final String stage =
AppConfig.getDomain().toLowerCase();
```

We first mine from the package and find that the code tokens ('`getDomain`', '`toLowerCase`') are strongly associated with high confidence (PMI = 13.2, appeared 10 times in the package). This gives

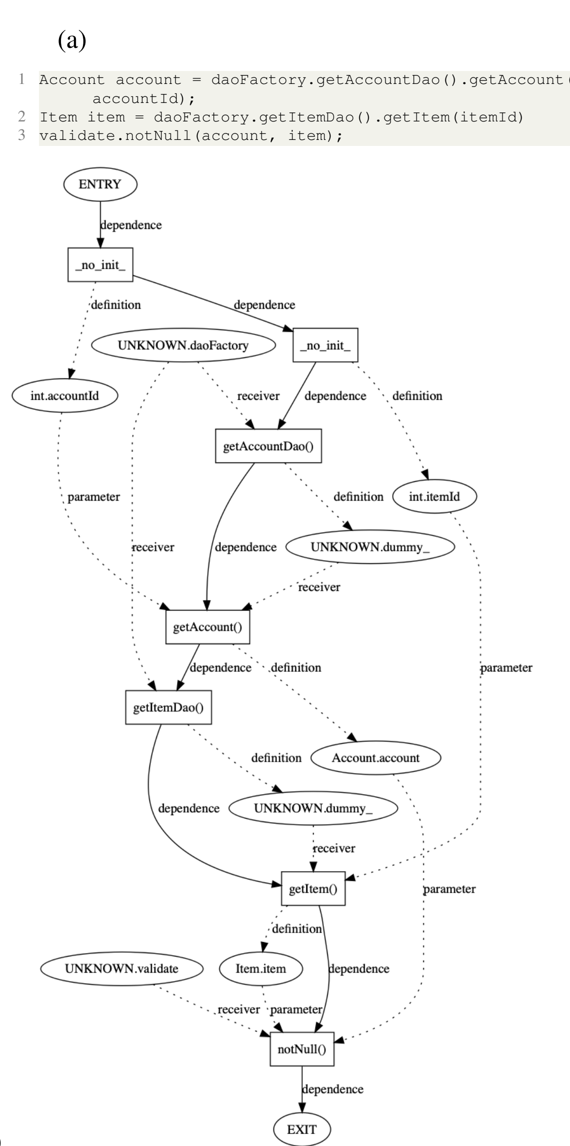

Figure 2: (a) Code snippet (b) Corresponding MU Graph.

an implicit rule that the token '`getDomain`' has to be immediately followed by '`toLowerCase`'. We then identify a code snippet that violates such pattern, where `getDomain()` is not followed by `toLowerCase()`. This is a potential inconsistency that leads to a missing API call. As a final step, we verify that the context of `getDomain()` in the abnormal code snippet is similar to the common patterns: both are initializing a string by getting the domain information from `AppConfig`. This results in a correct inconsistency detection that has been verified by the developer.

In the following subsections, we introduce the token level approach and graph level approach in more detail. The graph level approach is largely an extension of the token level approach, so the steps shared by both will be mostly introduced in the token level section.

B.1 Token level approach

In this approach, we take all the source code within a code repository as our data source. We use regular expression to tokenize source code and remove tokens with low frequencies. The code tokens maintain the same relative orders as in the source code. The *n-gram* of code tokens is defined in the same way of *n-gram* in texts. After the preprocessing we perform the following steps.

B.1.1 Mine code patterns

We collect the bigram of tokens with PMI higher than a pre-set threshold as our set of extracted code patterns. Alternatively, we can choose top bigrams ranked by their PMI values. Extracting bigrams with high PMI value will not only make the whole detection process more computationally efficient because of reduced candidate sets of bigrams, but also help reduce noises in the extracted code patterns, which may result in improved detection accuracy.

B.1.2 Detect abnormal code

We assign a conditional probability for each strongly associated bigram identified in the first step, defined by $p = N(a, b)/N(a)$, where $N(a, b)$ and $N(a)$ are the frequency of occurrences (known as *support*) of bigram (a, b) and token (a) individually. For instance, $p = 0.95$ indicates there is 95% chance that token a has to be followed by token b. We further reduce our bigram set by only keeping bigrams with the conditional probability higher than a threshold. To find the outliers, we retrieve the source code where a is not followed by b. As this is rare case, it is abnormal and a potential inconsistency. The same procedure works in another direction. We assign a conditional probability of $p = N(a, b)/N(b)$ and find the cases where b is not preceded by a as outliers. We have a lower bound of $N(a, b)$ to ensure the conditional probability is statistically significant.

B.1.3 Prune false positives

A violation of the common code patterns is not necessarily incorrect as the same code token may be used in various scenarios. After we obtained an outlier (a, x), where $x \neq b$, a key question to ask is: is (a, x) used in a similar way or similar context with (a, b) in common cases? If the context of (a, x) is functionally different than (a, b), it is a false violation and thus should not be regarded as inconsistent use of a.

For instance, here is an example of false positive.

violation:
```
private static List<Comment> filterAndSort(
list<Comment>unfilteredComments)
```
normal (11 times, shown 1):
```
return filterAndSort(unfilteredComments)
```

In this case, we found the token pair `(filterAndSort, unfilteredComments)` is strongly associated, because the method `filterAndSort` is called as many times as `filterAndSort(unfilteredComments)`. We found an outlier shown above that `filterAndSort` is not immediately followed by `unfilteredComments`. However, it is clear that the outlier is a method declaration clause and should not be in the same format as the method

call clause. We should remove such false violations by ensuring the context of the abnormal code is similar as the context of the code patterns. For simplicity, we define the context of a bigram as the line of code that contains the bigram. That says, the line of code that contains abnormal use of a token should be largely similar to the lines that normally use the token with the bigram. This forces us to look at not only the bigram but also the sequences of tokens around the bigram. We define 3 similarity measures to prune the false positives: a semantic similarity, token type similarity and code type similarity. Only when the violation passes all 3 similarity checkers it is determined as an inconsistency.

Semantic similarity Here, we regard the code snippets as plain texts and investigate the semantic similarity between the context of the abnormal code and the context of normal patterns. Intuitively, the more similar between those two, the more likely there is a true inconsistency given the abnormal code is different from the normal patterns. Empirically, we find this similarity relative to the similarity within the the contexts of normal patterns to be more helpful. Formally, for any abnormal code (i.e. bigram) i, we define the context of it as $c(i)$, and the set of all corresponding normal pattern occurrences in the whole corpus to be J. We define the relative similarity score (S-score) for i as:

$$\text{S-score}(i) = \frac{\frac{1}{|J|}\sum_{j \in J} Sim(c(i), c(j))}{\frac{1}{N}\sum_{j_1, j_2 \in J} Sim(c(j_1), c(j_2))} \tag{2}$$

where N is the number of times we randomly select j_1 and j_2 from J, and $Sim(c(i), c(j))$ is defined as the Jaccard similarity between $c(i)$ and $c(j)$, each being one line of code. The Jaccard similarity between any two sets A, B is defined as $Jaccard(A, B) = (|A \cap B|/|A \cup B|)$. We have a pre-set threshold of S-score(i) for the abnormal code i to pass this checker.

Token type similarity As a complement for the first similarity, we also check the similarity of token types in $c(i)$ and $c(j)$ as the token types will provide extra structural information on the code. We use a Java parser `javalang` to parse any line of Java code and retrieve the type of each token in it. For instance, after parsing the line of code `private static final String DOMAIN = AppConfig.getDomain()`, we obtained the type for each token: (Modifier)`private` (Modifier)`static` (Modifier)`final` (Identifier)`String` (Identifier)`DOMAIN` (Operator)= (Identifier)`AppConfig`.(Identifier)`getDomain()`. We define a similar S-score as Equation 2 but the Jaccard Similarity $Sim(c(i), c(j))$ is based on the type of each token instead of the value of each token. A threshold is also set to pass this checker.

Code type similarity The abnormal code should share similar intent or functionality as the normal cases to qualify as an inconsistency. We infer the high level type of a code snippet using the same `Javalang` parser. In the false positive example we discussed in Section 3.4, the abnormal code snippet is doing a method declaration while all the normal case snippets are doing a method call, thus the abnormal code is not an true inconsistency but a different use case of code token `filterAndSort`. Inferring such code type mismatch is a straightforward and efficient way to prune such false violations.

C SEQUENCE MINING-BASED INCONSISTENCY DETECTION

Extending the PMI-based bug detection to more than two tokens is fraught with scalability and data sparsity challenges: n-gram frequencies are not reliable as probability estimations when calculating PMI values for larger values of n. Hence we use sequence mining approach which is more scalable.

C.1 SEQUENCE MINING

The proposed approach consists of five steps: feature extraction, frequent itemset mining, association rule derivation, multiple sequence alignment, and contextual pruning. The following subsections explain the steps in detail.

C.1.1 Step 1: Extract features

We extract paths in MU Graphs as features. The types of paths extracted are listed in Table 1. Each type of path leads to detection of specific code bugs. The corresponding bugs are also indicated in the table.

For the *missing API* scenario, the extracted features from the MU Graph are nodes associated with API calls.

For the *missing null check scenario*, the feature extraction starts from the parameter or receiver edge between two API calls, and it also looks for additional edges coming out form the first API node and checks whether they are linked to any null-check related comparator or APIs.

For the *missing input validation* scenario, the feature extraction starts with the input parameter node in a method and follows its outgoing edges to related APIs that perform validation check on this parameter as well as the ones that consume this parameter. This not only includes the information of whether it is validated, but also how it is validated and how it is used.

The ordering of nodes in the path requires careful handling due to the complexity of MU Graph. The starting point is topological sort. However, the order given by topological sort is not necessarily aligned with the order of lines in the source code and this can cause confusions in the understanding of recommendations by developers. Therefore, we use a mixed sorting strategy. We firstly sort by line numbers, then for APIs on the same line, we sort them topologically.

Consider the listing and MU Graph shown in Figure 2. In this graph, there are two path sequences: `getAccountDao()->getAccount()->validate.notNull()` and `getItemDao()->getItem()->validate.notNull()`.
If we perform topological sort only, there are two possibilities:
`getAccountDao()->getAccount()->getItemDao() ->getItem()->validate.notNull()`
or
`getItemDao()->getItem()->getAccountDao() ->getAccount()->validate.notNull()`.
Instead, when we firstly sort by line numbers, and on the same line we determine API orders based on topological order, the API ordering is unique and what we want:
`getAccountDao()->getAccount()->getItemDao()-> getItem()->validate.notNull()`.

C.1.2 Step 2: Frequent Itemset Mining

Frequent patterns are patterns (itemsets, subsequences, or substructures etc.) that appear frequently in a dataset. Here we mine frequent patterns within the code repository under analysis. There are many choices for frequent itemset mining algorithms. In our implementation we use the Apriori algorithm Agrawal & Srikant (1994). Apriori is a seminal algorithm for mining frequent itemsets for Boolean association rules. Apriori employs an iterative approach known as a level-wise search, where k-itemsets are used to explore (k+1)-itemsets. The output of frequent itemsets mining are a group of frequent itemsets, i.e. "similar" sets (called "baskets") that contain many common items. In general, the larger the k, the fewer the number of baskets containing these k items. In our bug detection case, we observed the tendency that the larger the value of k, the more compelling the detection.

C.1.3 Step 3: Association Rule Derivation

Once the frequent itemsets have been found, it is straightforward to generate strong association rules from them, where strong association rules satisfy both minimum support and minimum confidence. The confidence of rule A=>B is defined as

$$confidence(A => B) = P(B|A) = \frac{support_count(A \cap B)}{support_count(A)} \tag{3}$$

where both A and B are sets of items (e.g. APIs). We find that the support and confidence measures are often insufficient for filtering out uninteresting association rules. To tackle this weakness, a correlation measure can be used to augment the support-confidence framework for association rule. There are various correlation measures, and we use the "lift" measure given below.

$$lift(A, B) = \frac{P(A \cap B)}{P(A) \times P(B)} \tag{4}$$

If $lift(A, B) < 1$, then the occurrence of A is negatively correlated with the occurrence of B, meaning that the occurrence of one likely leads to the absence of the other. If $lift(A, B) > 1$, then A and B are positively correlated, meaning that the occurrence of one implies the occurrence of the other. If $lift(A, B) = 1$, then A and B are independent and there is no correlation between them. *Lift and PMI are analogous. The difference is that in the PMI-based approach, A and B are single items, while in the case of lift, they are sets.*

C.1.4 Step 4: Sequence Alignment

The association rule gives us the findings of deviation instances from that rule. However, since the Apriori algorithm is based on set operation, ordering information of the items is lost during the process. Here, we want to enforce the ordering of items to identify the best supporting samples for the detection. In bug detection, explaining the findings to developers is as important as generating the findings. The baskets related to the same association rule A => B all share the same A part of the rule. However, items in A are only parts of those baskets and there are other items in the baskets. At the same time, the basket that misses the B part of the rule is the anomaly, i.e. the detection. We aim to find the basket most similar to the anomaly (but contains B) as our supporting example, which is included in our detection message to show the correct usage. We model this step as a sequence alignment process, namely, finding the best alignments for the items in part A for two sequences. This process not only produces the number of missing items in a sequence, but also their locations, which are used to filter the findings as well as generating the proper detection message.

C.2 An Illustrative Example

In this section, we will use an illustrative example to better explain the steps 1-4 presented above. The following code snippet is a Java method and we illustrate the process of find the missing API using this code snippet.

```
1  public void newConnectionError(Context context, long maxWaitMillis, Throwable cause) {
2     this.ConnectionDrop(context);
3     connectionErrorsMap.increment(context.getTrafficSource());
4     deck.increment(CounterType.NEW_CONNECTION_FAILED);
5     deck.increment(CounterType.SESSION_TRANSFER_FAILED);
6     LogEntry entry = EVENTS.error()
7        .append("NEW_CONNECTION_FAILED, SESSION_TRANSFER_FAILED, CLIENT_CONNECTION_DROP: src=")
8        .append(context.getSource())
9        .append(", cluster=")
10       .append(getClusterId(context))
11       .append(", id=")
12       .append(context.getConnectionId())
13       .append(". Failed to connect in ")
14       .append(maxWaitMillis)
15       .append(" ms");
16    addCauseIfNotNull(cause, entry);
17    entry.commit();
18 }
```

The first step is to build the MU Graph and extract paths, and since this is for missing API detection, the paths/features are the APIs called within this method. The resulting basket for this method is as follows:

```
1  ('EventListener.ConnectionDrop', 'Context.getSource', 'Map.increment', 'Deck.increment', 'Deck.increment', '
       Log.error', 'Context.getSource', 'EventListener.getClusterId', 'Context.getConnectionId', 'EventListener
       .addCauseIfNotNull', 'LogEntry.commit')
```

The second step is frequent itemset mining which groups the similar baskets that share many common APIs together, and Figure 3 shows one group containing 10 similar baskets. The third step is the rule derivation, and for this case, the following rule is derived.

```
1  A={'Deck.increment', 'LogEntry.commit', 'Log.error', 'Context.getConnectionId', 'EventListener.
       addCauseIfNotNull', 'EventListener.getClusterId'}
2  ==> B={'EventListener.ConnectionDrop'}
```

The support of the this rule is 10 as all 10 baskets in the group contain the APIs in A, and the confidence is 0.8 as there are 8 baskets contain the API in both A and B. The lift for this rule is 3.52 which is larger than 1, meaning that A and B are positively correlated. Figure 4 illustrates step 4, and all the matching APIs are color coded with the same color. It is clear that there is a missing API call 'EventListener.ConnectionDrop' at the first location.

Basket 0 : ('EventListener. ConnectionDrop', 'Deck.increment', 'Log.error', 'Context.getSource', 'EventListener.getClusterId', 'Context.getConnectionId', 'EventListener.addCauseIfNotNull', 'LogEntry.commit')
Basket 1 : ('EventListener. ConnectionDrop', 'Deck.increment', 'Deck.increment', 'Log.error', 'Context.getSource', 'EventListener.getClusterId', 'Context.getConnectionId', 'EventListener.addCauseIfNotNull', 'LogEntry.commit')
Basket 2 : ('EventListener. ConnectionDrop', 'Deck.increment', 'Deck.increment', 'Log.error', 'Context.getSource', 'EventListener.getClusterId', 'Context.getConnectionId', 'EventListener.addCauseIfNotNull', 'LogEntry.commit')
.
.
Basket 9 : ('Deck.increment', 'Log.error', 'Context.getSource', 'EventListener.getClusterId', 'Context.getConnectionId', 'EventListener.addCauseIfNotNull', 'LogEntry.commit') Group 1

Figure 3: Group of similar baskets after frequent itemset mining

('EventListener.ConnectionDrop', 'Deck.increment', 'Log.error', 'Context.getSource', 'EventListener.getClusterId', 'Context.getConnectionId', 'EventListener.addCauseIfNotNull', 'LogEntry.commit')

(MISSING, 'Deck.increment', 'Log.error', 'Context.getSource', 'EventListener.getClusterId', 'Context.getConnectionId', 'EventListener.addCauseIfNotNull', 'LogEntry.commit')

Figure 4: Sequence alignment to find matching and missing items

C.3 LENGTH-SUPPORT TRADEOFFS

The role of support is clear from the frequent itemset mining setup: the higher the support, the more "compelling" the rule. However, the role of the length of the "A" part of the rule is not as clear. Rule length and confidence are positively correlated. As a result, the required support could be reduced for longer sequences. We are interested in rules longer than a minimal length threshold. For these rules, we set different support constraints for different rule lengths. The longer the rule is, the less likely the paths/features within the rule co-occur by accident. Therefore, even if there are only a small number of such occurrences, the rule is still significant. In our bug detection application, longer rules usually indicate strong reasons to support such rule (e.g. a series of API calls in a method). Thus, only seeing a few examples is usually enough to derive such a rule.

To understand the length-support tradeoff better, let us compare the missing API detection using PMI-based and sequence based approaches. Although they are both detecting missing APIs, the underlying algorithms for them are different. Therefore, their detections usually do not overlap. For illustration purpose, consider a normal usage of API sequence <a>, <b>, <c>, <d>, <e>, and the API sequence under analysis is <a>, <c>, <d>, <e>. Clearly, there is a missing API <b> here that could be potentially problematic. The PMI-based approach looks at the bigrams <a>-<b> and <b>-<c>, while the sequence-based approach considers the entire <a> to <e> sequence. PMI-based requires a large support (or high PMI-score) to make the detection. The sequence-based does not require a support instead, because the entire sequence itself is a strong evidence due to its low probability of occurrence.

D REAL-WORLD EXAMPLES

We present a selection of sample findings to demonstrate the diversity of these detectors. The detections from PMI- and sequence-based detectors were complementary and there are almost no overlaps between the detections.

In the detection examples, some of the variable and API names have been changed to preserve confidentiality of the code.

Table 4: Token Level Inconsistency Detection Examples.

Code Issue	Detection

Missing API	**inconsistent**:
	return Base64.**getUrlEncoder().encodeToString**(bytes);
	normal (13 times, shown 1):
	return Base64.**getUrlEncoder().withoutPadding().encodeToString**(idbytes);

Messaging	**inconsistent**: error4display.append("**Odin User Materials** is a required field."); normal (20 times, shown 1): error4display.append("**User Odin Material** is a required field.");
Logging	**inconsistent**: **log.info** ("Missing {} from data {}", ID_FIELD, data); **normal** (47 times, shown 1): **log.error** ("Missing {} from data {}", ID_FIELD, data);
Typo	**inconsistent**: final String roleArn = String.format("arn:**aws:imm**::%s:role/Anonymous", **normal** (17 times, shown 1): final String roleArn = String.format("arn:**aws:iam**::%s:role/Anonymous",
Naming	**inconsistent**: OverlayUtil.renderPolygon(**g**,poly, color); **normal** (26 times, shown 1): OverlayUtil.renderPolygon(**graphics**,poly, LINE_OF_SIGHT_COLOR);

Visibility	**inconsistent**:
	public boolean equals(Object obj) {
	normal (26 times, shown 1):
	public boolean equals(**final** Object obj) {

Exception	**inconsistent**: throw new **InvalidArgumentException** ("Argument must be non-null"); **normal** (15 times, shown 1): throw new **IllegalArgumentException** ("Argument must be non-null");

Table 5: Graph Level Inconsistency Detection Examples.

Code Issue	Detection

General Missing API	**inconsistent**:
	protected static String getVersion(byte[] data) throws Exception {
	  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
	  DocumentBuilder builder = factory.newDocumentBuilder();
	  Document doc = builder.parse(new ByteArrayInputStream(data));
	normal (15 times, shown 1):
	private Element create() throws Exception {
	  DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
	  dbf.setNamespaceAware(true);
	  DocumentBuilder db = dbf.newDocumentBuilder();
	  Document doc = db.newDocument();

Missing Precondition	**inconsistent**:
	public static void tearDown() {
	  AppConfig.destroy();
	}
	normal (8 times, shown 1):
	public static void destroyAppConfig() {
	  if (AppConfig.isInitialized()){
	    AppConfig.destroy();
	  }
	}

Missing Logging	**inconsistent**:
	private StreamReader getStreamReader(final String stageLabel) {
	    return Context.getLabel(stageLabel).getStageVersion().getStreamReader();
	}
	normal (8 times, shown 1):
	private StreamReader getStreamReader(final String stageLabel) {
	   log.debug("Stage Version Label for doc config query : " + stageLabel);
	    return Context.getLabel(stageLabel).getStageVersion().getStreamReader();
	}

D.1 EXAMPLES OF PMI APPROACH

We show some sample detections for various subcategories of code issues from both token-level (Table 4) and graph-level approach (Table 5). In each example, we show the inconsistent piece of code (highlighted in bold) we detected as well as how the code snippet normally look like in similar cases.

D.2 EXAMPLES OF SEQUENCE-MINING APPROACH

D.2.1 SEQUENCE-BASED: NULL-CHECK

The following code snippet shows a detection made by null-check detector. Typically the output generated by the API 'System.getProperty' is checked against null and then consumed by other APIs such as 'log.info'. However, the variable 'ServerOverride' missed such check. We found it to be a copy paste error, namely, the developer used the 'ClientOverride' instead of 'ServerOverride' for the second null check in the code. The problematic code is highlighted with red background color while the corresponding correct code is highlighted with green background color. The same color coding is also applied for all the other examples.

```
1  String ClientOverride = System.getProperty(CLIENT_OVERRIDE_PROPERTY);
2
3    if (ClientOverride != null)  {
4      log.info("Found Client override {}", ClientOverride);
5      config.setUnitTestOverride(ConfigKeys.Client, ClientOverride);
6  }
7
8  String ServerOverride = System.getProperty(SERVER_OVERRIDE_PROPERTY);
9    if ( ClientOverride != null)  {
10
11     log.info("Found Server override {}", ServerOverride);
12     config.setUnitTestOverride(ConfigKeys.Server, ServerOverride);
13 }
```

D.2.2 SEQUENCE-BASED: TRY-EXCEPTION

The following code snippets show a detection on the try-exception handling. The problematic code snippet is shown as follows

```
1  try {
2     if (null != PriceString) {
3         final Double Price = Double.parseDouble(PriceString);
4         if (Price.isNaN() || Price <= 0) {
5             throw new  InvalidParameterException (
6             ExceptionUtil.getExceptionMessageInvalidParam(PRICE_PARAM));
7         }
8     }
9  } catch (NumberFormatException ex) {
10        throw new  InvalidParameterException (
11        ExceptionUtil.getExceptionMessageInvalidParam(PRICE_PARAM));
12 }
```

The tool made the following recommendation: "We found 4 occurrences of handling an exception thrown by 'Double.parseDouble' in your code. We recommend using catch with 'InvalidPriceConfigException()' or an equivalent exception that provides the same level of detail. The selected code shows an occurrence in which this does not happen. The following is an example in your code of using catch to provide the recommended detail: "

```
1  try {
2      final Double Price = Double.parseDouble(launchTemplateOverrides.getPrice());
3      if (Price.isNaN() || spotPrice <= 0) {
4          throw new  InvalidPriceConfigException (
5             ExceptionUtil.getExceptionMessageInvalidParam(PRICE_PARAM));
6      }
7  } catch (NumberFormatException ex) {
8      throw new  InvalidPriceConfigException (
9          ExceptionUtil.getExceptionMessageInvalidParam(PRICE_PARAM));
10 }
```

Basically, there is a paticular exception (InvalidPriceConfigException) designed for this logic. The developer confirmed that it should be fixed and also made the change.

D.2.3 SEQUENCE-BASED: API CALLS

The following code snippets show a detection on missing API (API misuse in this case). The problematic code snippet is shown as follows.

```
1  .fold(error -> {
2       System.out.println
3      ("Unknown error occurred while executing [context={}]" + error);
4      if (retryableException.test(error)) {
5         return Outcome.<Work, State>builder()
6      }
```

The tool made the following recommendation: “We found 7 occurrences of methods similar to the method in the selected code that call ‘Logger.error’. However, the selected method does not make that call. We recommend adding a call to it if this is not intentional. The following is an example of a similar method in your code that makes this call: "

```
.fold(error -> {
    log.error
    ("Unknown error occurred while executing [context={}]", context, error);
    if (retryableException.test(error)) {
       return Outcome.<Work, State>builder()
    }
```

Basically, these are very similar methods, and in normal cases the error is logged by using logger.error, but in this case the developer uses printout. Using the logger is usually a better practice as it provide additional information.