Sequence-based Program Semantic Rule Mining and Violation Detection

Abstract

In software development, source code that violates semantic rules typically has performance or functional defects, but can still compile or run normally. Therefore, it is a difficult challenge to detect program defects caused by violations of semantic rules. Some existing research has adopted itemset-based rule mining and detection methods, but these methods have significant room for improvement in the number of detected defects and accuracy, due to the poor combination of source code order information and control flow information. To better detect such defects, this report proposes a sequence-based program semantic rule extraction and violation detection method called SPUME (Sequence-based Rule Mining and Enforcing). This method converts program source code into an intermediate representation sequence, and then uses sequence rule mining algorithms to extract semantic rules from it, and detects defects in the source code based on these semantic rules. To verify the effectiveness of SPUME, the method was applied to seven different open-source projects of different scales and types, and compared with three baseline methods, including PR-Miner, Tikanga, and Bugram. The experimental results show that compared to PR-Miner, which is based on unordered itemset mining, and Tikanga, which is based on graph models, SPUME has significantly improved detection performance, speed, and accuracy. Compared to the Bugram method based on Ngram language models, SPUME detects more program defects more efficiently while maintaining similar accuracy.

Keywords: Semantic Rule Mining; Overlapping Clustering; Defect Detection

1 Introduction

As programming languages have evolved, their functionality and variety have become increasingly rich in response to the growing complexity of software system development needs. Despite the diverse syntax rules of different programming languages, developers can still find detailed descriptions in their language specifications and write code that fully complies with the syntax rules of the programming language with the help of various syntax checking and auto-completion tools. However, in actual development, in addition to syntax rules, programmers often need to follow numerous semantic rules. For example, when using manual memory management languages, they should call the corresponding free function at the appropriate time after calling the alloc function to allocate memory to avoid memory leaks. Similarly, when writing concurrent programs, they should use the lock function to lock critical resources and call the unlock function after accessing them to ensure the normal operation of the entire software system.

```
//openssl/crypto/pem/pem_lib.c
int PEM_read_bio_ex(BIO *bp, char **name_out,
                    char **header,unsigned char **data,
                    long *len_out, unsigned int flags){

    BIO *headerB = NULL, *dataB = NULL;
    ...
    dataB = BIO_new(bmeth);
    ...
    if (BIO_read(dataB, *data, len) != len){
    ...
    BIO_free(dataB);
    ...
}
```

```
//openssl/crypto/pem/pem_lib.c
void *PEM_ASN1_read(d2i_of_void *d2i, const char *name,
                    FILE *fp, void **x,
                    pem_password_cb *cb, void *u){

    BIO *b;
      ...
    if ((b = BIO_new(BIO_s_file())) == NULL) {
      ...
    BIO_set_fp(b, fp, BIO_NOCLOSE);
    ret = PEM_ASN1_read_bio(d2i, name, b, x, cb, u);
    ...
    BIO_free(b);
    ...
}
```

Although developers may notice some simple and common semantic rules through their own development experience, the semantic rules that need to be followed in actual software development are often more specific and complex. As shown in Figure 1, in the OpenSSL library, I/O-related operations are abstracted as a BIO type to hide underlying details. Before using methods like BIO_read to retrieve data from a BIO object, it is necessary to call BIO_new to initialize the BIO object and call BIO_free to release the memory it occupies when ensuring that a BIO object is no longer in use. In addition, as shown in Figure 2, when the I/O type is a special type, such as file I/O, additional settings need to be made to the BIO object by calling functions like BIO_set_fp after BIO_new to make the program work properly. In the OpenSSL source code, there are over 450 occurrences of semantic rules related to BIO objects, and similar semantic rules exist in any complex system. These semantic rules usually consist of multiple elements, including variable declarations, function calls, and even control flow information, which imply the inherent logic of specific programs. Violating these semantic rules will affect the program's running efficiency and may even lead to functional defects. At the same time, due to the large scale and strong coupling with specific projects, it is difficult to summarize such semantic rules into some common form, which makes it difficult to form documents or develop detection or repair tools for them.

To address this dilemma, some work has been proposed to extract semantic rules from program source code and apply them to defect detection [1-6,23]. Li et al. proposed the PR-Miner tool, which uses frequent itemset mining [11] to find frequently occurring variable and function calls. This method can effectively construct semantic rules, but the itemset-based approach will lose the order information in the data, leading to a higher false positive rate. Wasylkowski et al. proposed the JADET [4] and TIKANGA [5] tools, which use a combination of frequent itemset mining and graph models to discover abnormal object usage and find the prerequisites required when specific call patterns appear. However, due to the limited form of rules discovered and the need to work with compiled Java bytecode, they have significant limitations. Song et al. proposed the Bugram [6] method, which uses the Ngram [7] natural language processing model to find low-probability patterns and detect program defects. Compared to previous work, Bugram has higher accuracy and can directly operate on program source code. However, due to its dependence on the Ngram method, it has poor interpretability and higher requirements for computational and training data.

Based on the above issues, we conducted research on the extraction methods of program semantic rules and proposed the SPUME (Sequence-based Programming rUle Mining and Enforcing) method to mine program semantic rules. This method converts program source code into a higher-level intermediate representation and applies sequence rule mining methods to extract semantic rules. Compared to previous work, SPUME discovers semantic rules with higher accuracy and richer forms, which are not limited to function or programming interface calls but also involve control flow information. In addition, SPUME introduces a data processing method based on overlapping clustering, which greatly improves the efficiency advantage of this method and makes it suitable for mining work of various scales. For the semantic rules discovered, SPUME also provides efficient detection algorithms corresponding to them to detect behaviors in program source code that violate semantic rules.

2 Sequence Rule Mining

Association rule mining [12] can discover interesting relationships between data and has been widely used in program source code mining. Traditional association rule mining methods are usually applied to set data, while program source code data is a type of sequential data. Therefore, applying methods designed for unordered sets to mine semantic rules may lead to certain defects. Therefore, in this report, we will use sequence-based rule mining algorithms [15] to ensure that the order of elements is preserved.

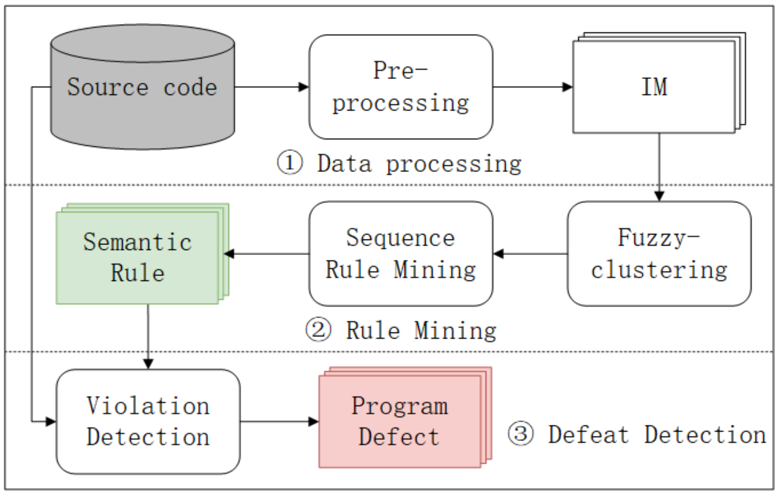
Regarding the description of certain concepts in sequence rules, there may be differences in different algorithm implementations. To adapt to specific tasks, we redefine them as follows. A sequence database is a collection of sequences

$S = \{s_1, s_2, ..., s_n\}$ and a set of items $I = \{i_1, i_2, ..., i_n\}$. An arbitrary sequence is defined as $s_x = \langle x_1, x_2, ..., x_n \rangle$, where x_1, x_2 belong to I. A sequence rule in the form of $X \Rightarrow Y$ describes the relationship between sequence X and sequence Y, where both X and Y have a length greater than 0 and do not overlap with each other. The meaning of the rule $X \Rightarrow Y$ is that when X appears as a subsequence in some sequence, Y appears in that sequence at a position after X. In sequence rules, X and Y are respectively referred to as the left-hand side (LHS) and right-hand side (RHS). Multiple metrics derived from association rules, such as support, confidence, and lift, are used to measure sequence rules. In this report, we mainly use support, confidence, and lift. Among them, support is defined as $sup(X \Rightarrow Y) = sup(X,Y) / |s|$, confidence as $conf(X \Rightarrow Y) = sup(X,Y) / sup(X)$, and lift as $lift(X \Rightarrow Y) = sup(X,Y) / (sup(X) * sup(Y))$. $sup(X)$ represents how many sequences X appears in, and $sup(X,Y)$ represents how many sequences X and Y appear in order.

In this report, we will use an improved sequence rule mining algorithm called RuleGrowth [15] to complete the mining task. Unlike the traditional "candidate generation-testing" method [14], RuleGrowth uses a "growth" method similar to the PrefixSpan [16] algorithm, which is based on legal sequence rules between pairs of items to derive other rules. It also uses multiple strategies to avoid global scanning of the sequence database, greatly optimizing the algorithm efficiency. RuleGrowth is currently one of the best-performing algorithms in this field and has been used in some of the latest data mining work.

3 SPUME

Figure 3 shows an overview of the SPUME workflow, which mainly consists of three modules: data processing, rule mining, and violation detection.



1. In the data processing module, SPUME first parses the program source code and serializes it. Then, combined with a series of heuristic strategies, the source code sequence is optimized and transformed into an intermediate representation that is more suitable for sequence rule mining methods.

2. In the rule mining module, SPUME uses the overlapping clustering algorithm to derive multiple data clusters from the original sequence data set that may contain similar semantic rules. Each data cluster is then used as a sequence database for sequence rule mining algorithms to obtain semantic rules.
3. In the violation detection module, SPUME will use the obtained semantic rules to perform violation detection on the source code data. In this part, SPUME proposes an efficient violation detection scheme based on state transition for this purpose.

3.1 Date Processing

Untreated program source code is difficult to apply directly to data mining tasks. Therefore, in this stage, SPUME proposes a multi-stage data processing method to transform program source code into an intermediate representation suitable for sequence rule mining. Specifically, this method includes three parts: source code serialization, function call expansion, and intermediate representation construction. In this report, SPUME will use a C language project as an example to introduce the semantic rule mining method. However, it is worth emphasizing that the main structure of this data processing method is universal and can be applied to different programming languages with appropriate adaptations.

3.1.1 Source Code Serialization

In C language, a function is a code block that can be reused to perform a specific independent function. Therefore, functions can usually be regarded as the smallest unit containing complete semantic rules. In this report, the basic method used by SPUME to process C language source code is to transform functions in the source code into an intermediate representation sequence and form a sequence database for subsequent mining work.

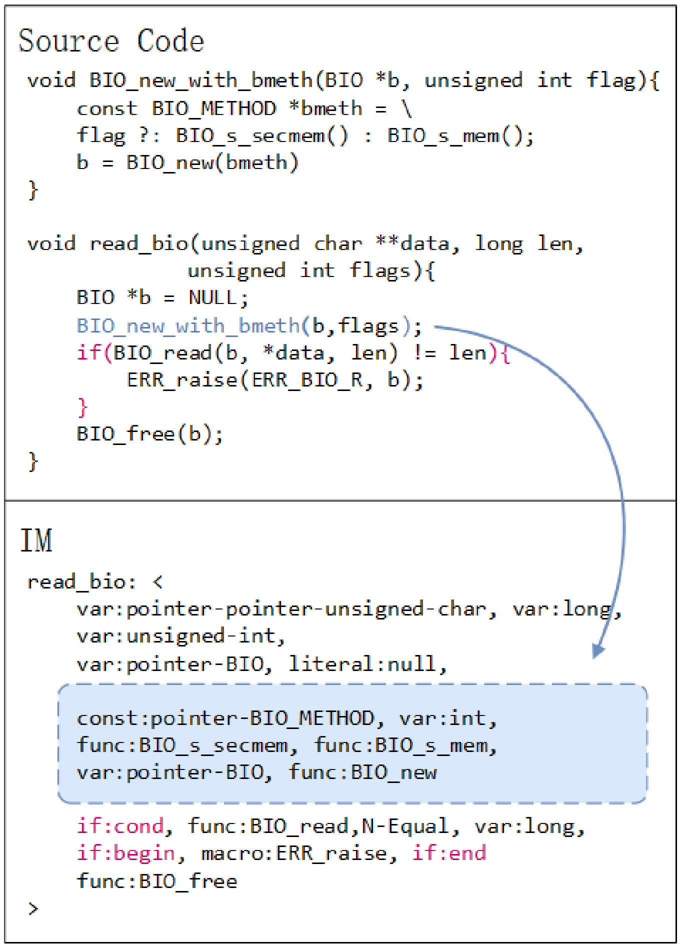
To parse C language source code, we use the open-source tool PyCParser to process the source code, including expanding header files, removing comments, analyzing tokens, and identifying syntax structures, and outputting a list of functions and their corresponding abstract syntax trees (ASTs). The AST is an abstract representation of the syntax structure of the source code. Each node in the AST contains information such as the name and type of its corresponding token. In particular, in this step, we do not expand "macros" but parse them as syntax tree nodes similar to function calls. After obtaining the AST, we perform a pre-order depth-first traversal of the tree to obtain a node sequence for each function.

3.1.2 Function Call Expansion

C language supports nested function calls, so when analyzing the semantic rules contained in a function, we should consider all the function call chains it contains. As shown in Figure 2, although the function read_bio contains the semantic rules about the BIO object mentioned in the introduction, the function BIO_new, as part of the semantic rules, appears in the helper function BIO_new_with_bmeth called by read_bio. Therefore, if we directly use the node sequence corresponding to each function's syntax tree to generate the intermediate representation, we will miss such rules.

One intuitive method to solve this problem is to expand the functions at their call sites , i.e., replacing the function call nodes in the syntax tree sequence with their corresponding sequences. However, if the expansion depth is not limited, the resulting sequence may become too long, which can have a negative impact on the efficiency of subsequent rule mining algorithms. Therefore, determining the boundary of function expansion is an important issue in this process. In C language, developers usually organize functions that are closely related to each other in the same file. When a called function is not in the same file as its caller, it is more likely to belong to another functional module or third-party library, and the possibility of containing semantic rules in such function calls is also smaller. Therefore, to balance mining effectiveness and efficiency, SPUME only analyzes and expands function call chains that belong to the same file.

3.1.3 Intermediate Representation Construction


Source Code

```
void BIO_new_with_bmeth(BIO *b, unsigned int flag){
    const BIO_METHOD *bmeth = \
    flag ?: BIO_s_secmem() : BIO_s_mem();
    b = BIO_new(bmeth)
}

void read_bio(unsigned char **data, long len,
             unsigned int flags){
    BIO *b = NULL;
    BIO_new_with_bmeth(b,flags);
    if(BIO_read(b, *data, len) != len){
        ERR_raise(ERR_BIO_R, b);
    }
    BIO_free(b);
}
```

IM

```
read_bio: <
    var:pointer-pointer-unsigned-char, var:long,
    var:unsigned-int,
    var:pointer-BIO, literal:null,

    const:pointer-BIO_METHOD, var:int,
    func:BIO_s_secmem, func:BIO_s_mem,
    var:pointer-BIO, func:BIO_new

    if:cond, func:BIO_read,N-Equal, var:long,
    if:begin, macro:ERR_raise, if:end
    func:BIO_free
>
```

The main goal of this stage is to construct an intermediate representation suitable for mining algorithms based on the node sequence of the structured syntax analysis tree. Although we have filtered out noise that may have a negative impact on mining algorithms, such as semicolons and comments, in previous steps, we still need to further abstract the remaining data to shield some unnecessary details when constructing the intermediate representation. To achieve this, SPUME has developed a set of rules to convert syntax tree nodes into one or more text representations in the form of "token category: token abstraction". The rules are as follows.

For variables and constants that contain only a single token, we use "val" and "const" as their token categories, and use the name of their data type as their token abstraction. For example, in Figure 4, variable "b" has an actual type of a pointer to the BIO object, so it is represented as "var:Pointer-BIO".

For literals, we use "literal" as their token category, and match their token abstraction to a string "string", integer "integer", or null value "null" based on regular expressions.

For function call nodes and macro nodes, we set their token categories as "func" and "macro", respectively, and use their names as their token abstractions. For example, in Figure 4, the call to the BIO_read function is represented as "func:BIO_read", and the use of the ERR_raise macro is represented as "macro:ERR_raise". For their related parameters, we do not need to abstract them because these details are guaranteed by the compiler, and what we care about are higher-level semantic rules.

For control flow-related nodes, we use different representations based on their specific features. For conditional control flow statements like "if(...){...}else{...}", we represent them as "if:cond, ... , if:begin, ... , if:end, else:begin, ... else:end". This abstraction method can fully leverage the advantages of sequence rule mining methods. For example, if we find that a part of a rule in the mining result is wrapped between "if:cond" and "if:begin", we can conclude that this part belongs to one of the if statement's conditions. In addition, we also make corresponding abstractions for control flows like for/do/while.

Constructing an intermediate representation is crucial for subsequent mining work. Since the mining algorithm uses the support of rules as an important metric, we need to shield unnecessary details so that semantic rules can appear frequently in the data to be mined in the same abstract form.

3.2 Rule Mining

In this stage, the main task of SRMVD is to mine frequent sequences from the intermediate representation of the source code and construct semantic rules based on them.

3.2.1 Overlapping Clustering

Large software systems typically consist of multiple functional modules, and some semantic rules may appear frequently in certain modules but rarely in others. Therefore, if we directly apply the mining algorithm to the entire software source code data, such semantic rules will be “diluted” by data from other modules and difficult to discover. To address this issue, in this stage, SPUME aggregates function sequences that may contain similar semantic rules into data clusters and applies subsequent mining algorithms to each data cluster separately. Clustering algorithms can partition data into multiple clusters containing similar data based on a given similarity measure, and the clusters do not intersect with each other. However, it should be noted that a function may contain multiple semantic rules, and its sequence may belong to multiple data clusters. Therefore, SPUME uses the overlapping clustering algorithm Fuzzy-DBSCAN to complete this task. Fuzzy-DBSCAN is based on the hierarchical clustering algorithm DBSCAN, with the addition of fuzzy kernel expansion and fuzzy boundary expansion to aggregate overlapping data clusters. In terms of similarity measurement, SPUME uses the CodeBERT pre-trained model based on the Transformer architecture, which is designed for both programming language and natural language processing, to convert the intermediate representation of function sequences into semantic vectors, and uses cosine similarity as the similarity measure.

3.2.2 Sequence Rule Mining and Filtering

In this section, we will mine sequence rules and filter out semantic rules from them. SPUME considers two basic types of semantic rules to be mined, which are backward sequence rules $X \Rightarrow Y$ and forward sequence rules $X \Leftarrow Y$. The definition of backward sequence rules is consistent with the sequence rules introduced in Section 2 of this report. The forward sequence rule $X \Leftarrow Y$ is defined as follows: when Y appears as a subsequence in some sequence, X appears in that sequence at a position before Y. The definitions of support, confidence, and lift for forward sequence rules can be analogized to those for backward sequence rules.

We use the Rulegrowth algorithm to mine backward sequence rules. The Rulegrowth algorithm takes a sequence database SDB, a minimum support threshold $minsup$, and a minimum confidence threshold $minconf$ as parameters and outputs all backward sequence rules in the sequence database with support and confidence higher than the given thresholds. For example, we define sequences X_1, Y_1, and Y_2 as shown in Table 1. In a data cluster containing 25 sequences, sequence X_1 appears in 20 sequences, and after it, sequences Y_1 and Y_2 appear 15 and 19 times, respectively. Therefore, the support of sequence rule R_1: $X_1 \Rightarrow Y_1$ and rule R_2: $X_1 \Rightarrow Y_2$ are 0.6 and 0.76, and the confidence is 0.75 and 0.95, respectively. When we set the minimum support threshold to 0.5 and the minimum confidence threshold to 0.9, although both R_1 and R_2 satisfy the support requirement, the former is eliminated due to not meeting the confidence requirement. The underlying reason is that the BIO_read function is only one of many operations of the BIO object, so the confidence of rule R_1 is lower than the actual semantic rule R_2.

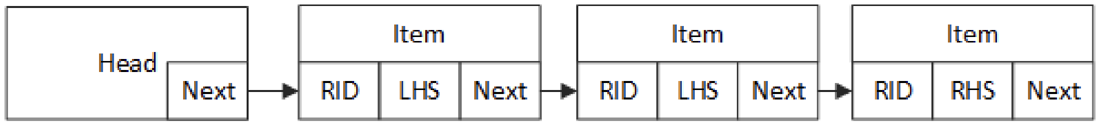
X_1	<var:Pointer-BOP, func:BIO_new>
Y_1	<func:BIO_read>
Y_2	<func:BIO_read>

However, evaluating rules solely based on support and confidence may lead to misjudgments. For example, in the testing module of a software, the log function is used at the end of almost every function to store test results. Therefore, when the sequence <func:log> appears as the "right-hand side" of a backward sequence rule, the rule may have both high support and high confidence, but in reality, calling the log function at the end of any function is not a semantic rule. Based on this situation, we have improved the Rulegrowth algorithm by adding lift calculation in addition to confidence calculation. When Y in the sequence rule $X \Rightarrow Y$ appears too frequently in the dataset, the lift of the rule will decrease. Therefore, incorporating lift as an evaluation metric can avoid mistaking such sequence rules as semantic rules.

For mining forward sequence rules, we use the following method. For a sequence database SDB, we reverse all sequences in it to obtain a new database SDB', and use it as the input for Rulegrowth to obtain a series of backward sequence rules. For any backward rule R': $X' \Rightarrow Y'$, we reverse the sequences X' and Y' to obtain new sequences X and Y, and $X \Leftarrow Y$ is one of the forward sequence rules in the sequence database SDB with the same support, confidence, and lift as R'. Based on this method, we can directly use the Rulegrowth algorithm to mine and filter forward sequence rules without implementing new algorithms.

3.3 Defect detection

For the violation detection of a single sequence rule, it can be accomplished in linear time through a naive sequence matching method. However, as the number of rules increases, the time required for detecting violations also increases exponentially. In this report, we propose a state transition-based rule violation detection algorithm (STRVD) that can efficiently detect violations of any rule in the source code. The algorithm for detecting violations of backward sequence rules is shown in Algorithm 1, while the detection process for forward rules is similar to the mining process and can be implemented using the algorithm for detecting backward rules, so it is not further described.



For any backward sequence rule, we convert it into a linked list format as shown in Figure 5 for storage. Each node in the linked list corresponds to an item in the rule and contains not only the data field and pointer field but also the identifier of the rule and the position information of the node in the rule. At the beginning of the algorithm, we take the linked list form of the rule and the source code to be detected as input, and convert the source code to an intermediate representation. For each rule, we generate a pointer pointing to its head node and categorize it based on the item corresponding to its next node. Then, we traverse each item in the intermediate representation of the source code, advance all pointers in the category corresponding to the current item, and categorize these pointers again in the same way. When the traversal is complete, we check the situation of all pointers. If a pointer stops at the node corresponding to the "right-hand side" area of a rule, the program source code is considered to violate the semantic rule.

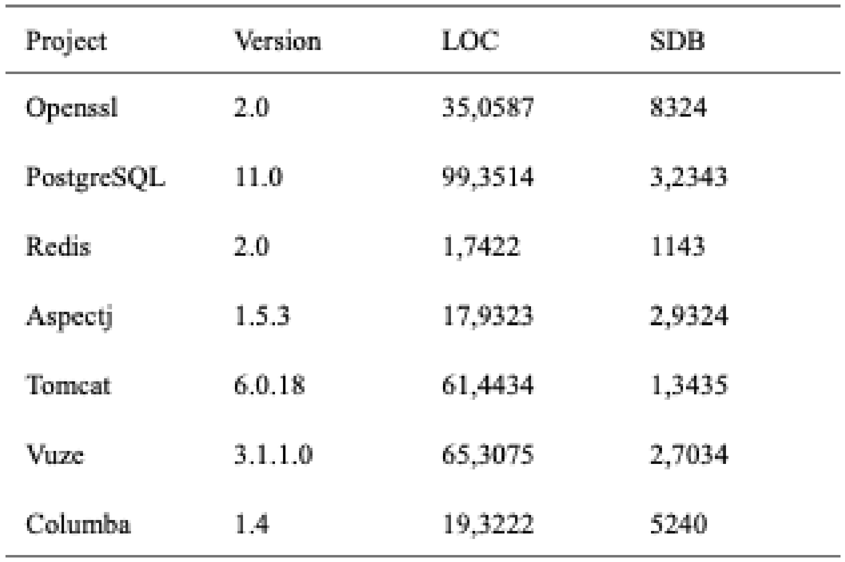
```
Algorithm1 STRVD
Input：Src, RLinklist
Output：Viloations
1.  IR = Process(Src)
2.  Set m as a hash table with (k, v) as (item, set)
3.  foreach RL in LinklistRules
4.        pointer = RL.head
5.        put pointer in M[pointer.next.item]
6.  for item in IR:
7.        for pointer in M[item]
8.               remove pointer from M[item]
9.               if pointer.next is not null:
10.                    pointer = pointer.next
11.                    put pointer in M[pointer.next.item]
12. for set in M.values:
13.       for pointer in set:
14.             if pointer.posi is Y:
15.                    put pointer.RuleID in Violations
```

4 Experimental Results and Analysis

This section will demonstrate the feasibility and effectiveness of the SPUME method for mining semantic rules and detecting defects in source code data through experimental design.

4.1 Experimental Design

In the actual implementation, we adapted the data processing module of the SPUME method for both C and Java languages. Therefore, we selected a series of well-known C and Java language projects to construct experimental data, which are widely used in the real world. After obtaining the source code of the projects, we processed the source code data using the data processing module of the SPUME method to obtain the intermediate representation required by the SPUME method and form the sequence database to be mined. The specific statistical information of the dataset is shown in the following table, where LOC and SDB represent the code lines and the size of the sequence database of the projects, respectively.



Project	Version	LOC	SDB
Openssl	2.0	35,0587	8324
PostgreSQL	11.0	99,3514	3,2343
Redis	2.0	1,7422	1143
Aspectj	1.5.3	17,9323	2,9324
Tomcat	6.0.18	61,4434	1,3435
Vuze	3.1.1.0	65,3075	2,7034
Columba	1.4	19,3222	5240

For the parameter setting of the Fuzzy-DBSCAN algorithm in the SPUME method, we dynamically generate the parameters using a multi-objective optimization algorithm based on the actual dataset. Therefore, the SPUME method only needs to input the minimum support threshold $minsup$, the minimum confidence threshold $minconf$, and the minimum lift threshold $minlift$. By default, we set them to 0.7, 0.9, and 1, respectively.

4.2 Research Questions and Results Analysis

The experiments in this report will mainly answer the following three research questions:

1. Can the data cluster generation strategy based on overlapping clustering effectively optimize the effectiveness of semantic rule mining?
2. Can SPUME detect defects in source code based on the mined semantic rules?
3. How does the performance of the SPUME method compare in terms of overall efficiency?

4.2.1 Optimization based on overlapping clustering

The purpose of using the Fuzzy-DBSCAN algorithm in SPUME is to cluster data that may contain similar semantic rules, which can reduce the size of the dataset and improve the mining effectiveness. To verify the effectiveness of this approach, we first implemented the SPUME method without overlapping clustering optimization, which we refer to as SPUME-w/o-FD.

To compare the effectiveness of the two methods, we applied them to the sequence databases of each project with default parameters and obtained the required running time for rule mining (excluding data preprocessing time) and the mining results. Due to the large number of mining results, it is difficult to verify them one by one. Therefore, we extracted 30 rules that meet the Pareto front or approximate front based on confidence and lift from the mining results and counted them as Rule@30. We invited experts in the field to evaluate whether these rules are true semantic rules. The specific experimental results are shown in Table 3.

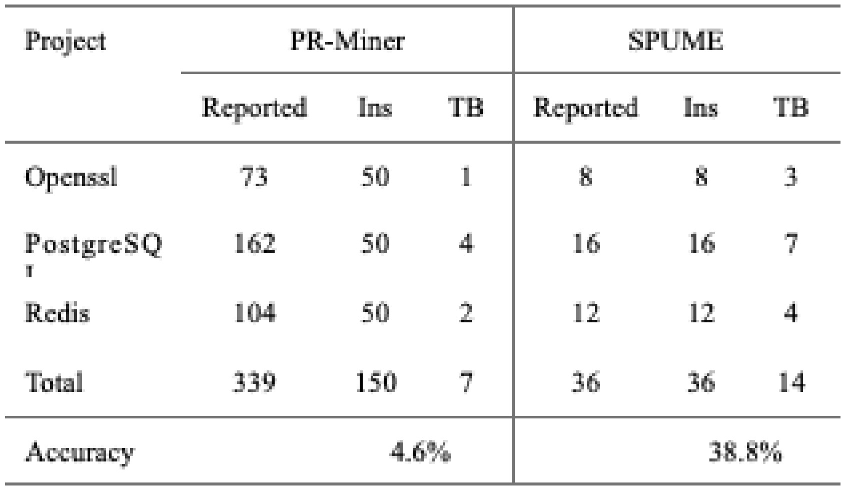
Project	SDB	SPUME-w/o-FD		SPUME	
		Time	Rule@30	Time	Rule@30
Redis	1143	4	13	2	27
Columba	5240	15	16	9	22
Openssl	8324	23	11	8	25
Tomcat	1,3435	39	9	11	28
Vuze	2,7034	91	14	43	28
AspectJ	2,9324	108	12	46	23
PostgreSQL	3,2343	143	18	45	27
Total		420	93	174	180

It can be observed that as the data size increases, the required mining time of SPUME-w/o-FD increases rapidly, while for the SPUME method, the use of overlapping clustering optimization allows it to control the dataset size used in a single mining, resulting in a nearly linear growth trend in the required time. Overall, SPUME is on average only 41.4% of the time required by SPUME-w/o-FD method, but its mining results are significantly better than SPUME-w/o-FD. Among the 30 high-confidence and high-lift sequence rules extracted, SPUME discovered an average of 93.5% more semantic rules than SPUME-w/o-FD. The experimental results demonstrate that the overlapping clustering method can generate small, semantically cohesive data clusters, which have a significant positive effect on the efficiency and results of subsequent mining.

4.2.2 The defect detection effectiveness of SPUME

To verify the defect detection effectiveness of SPUME, we selected PR-Miner, Tikanga, and Bugram as baseline works for comparison with SPUME.

As PR-Miner did not publicly release its tool prototype, in this experiment, we re-implemented its model based on its report description and selected the optimal parameter settings according to the experiment. In addition, PR-Miner can only be applied to C language, so we compared the corresponding version of SPUME with PR-Miner on the C language dataset, and the results are shown in Table 4.



Project	PR-Miner			SPUME		
	Reported	Ins	TB	Reported	Ins	TB
Openssl	73	50	1	8	8	3
PostgreSQL	162	50	4	16	16	7
Redis	104	50	2	12	12	4
Total	339	150	7	36	36	14
Accuracy	4.6%			38.8%		

Due to the excessive number of reported program defects by PR-Miner, we checked the optimal results based on its original metric standards. Ins and TB in Table 4 respectively represent the number of defects checked manually and the number of confirmed real defects. From the comparison results, it can be seen that under the setting of selecting the optimal 50 defect reports, SPUME can discover more real program defects than PR-Miner, and is significantly better than PR-Miner in terms of accuracy.

Tikanga and Bugram were applied to Java project datasets in their original experiments and published corresponding evaluation data. Therefore, we used the Java-adapted SPUME as a baseline work to compare with them on the same dataset. To ensure fairness of the comparison, we used the same experimental settings as much as possible when running SPUME, including using default parameters for mining and detection, and selecting the top 25% defect reports for manual evaluation from the results, etc. Table 5shows the corresponding experimental results. From the table, it can be observed that although the accuracy of SPUME is 0.6% lower than Bugram, it detected 7 more program defects. Compared with Tikanga, SPUME detected 2 additional program defects and had a 5.6% improvement in accuracy.

Project	Tikanga		Bugram		SPUME	
	Reported	TB	Reported	TB	Reported	TB
AspectJ	42	9	5	0	49	8
Tomacat	3	0	12	4	7	2
Vuze	56	0	8	0	19	0
Columba	5	1	6	1	5	2
Total	106	10	32	5	80	12
Accuracy	9.4&		15.6%		15%	

4.2.3 The overall efficiency of SPUME

In 4.2.1, we verified the positive impact of overlapping clustering optimization on mining efficiency, while in this question, we will focus on the overall efficiency of SPUME. We first recorded the time required for SPUME in the three stages of data processing, rule mining, and defect detection, and calculated the total time required. For comparison, we applied Tikanga and Bugram to the same dataset and compared their total time required with SPUME, as shown in Table 6. The columns P, M, and D respectively represent the time required for data processing, rule mining, and defect detection.

Project	SPUME				Tikanga	Bugram
	P(s)	M(s)	D(s)	Total(s)	Total(s)	Total(s)
AspectJ	122	46	25	193	248	319
Tomcat	58	11	8	77	98	114
Vuze	132	43	19	194	246	332
Columba	26	9	5	40	48	63
Total	338	109	57	504	640	828

The results show that compared with Tikanga and Bugram, SPUME requires significantly less time for the overall process, mainly concentrated in the data processing task (67%). Under the optimization of the overlapping clustering method and the fast rule detection algorithm, the rule mining and defect detection modules of SPUME already have very high efficiency.

Conclusion

In this report, we proposed a sequence-based program semantic rule extraction and defect detection method called SPUME. This method first converts the source code into multiple intermediate representation data clusters with small size and high semantic rule cohesion, and then uses a sequence rule mining algorithm to construct semantic rules containing sequence and control flow information, and detects program defects based on these rules. In the experiment, we applied SPUME to seven open-source projects of different types and sizes, and compared it with PR-Miner, Tikanga, and Bugram. The results show that compared with PR-Miner based on unordered itemset mining and Tikanga that only considers partial sequence information, SPUME has significantly improved in the number and accuracy of defect detection. Compared with Bugram, which is based on the Ngram language model and combines sequence and control flow information, SPUME detected more program defects while maintaining similar accuracy. Therefore, SPUME can be used as a replacement or supplement to existing program defect detection methods. Next, we will optimize the data processing method and expand the form of rules that can be mined based on existing work to further improve the effectiveness of defect detection.

Reference

[1] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. SIGSOFT Softw. Eng. Notes 30, 5 (September 2005), 306–315.
[2] L. Benjamin and T. Zimmermann. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In FSE' 05, pages 296–305, 2005.
[3] R.-Y. Chang, A. Podgurski, and J. Yang. Finding What's Not There: A New Approach to Revealing Neglected Conditions in Software. In ISSTA' 07, pages 163–173, 2007.
[4] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting object usage anomalies. In Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE '07). Association for Computing Machinery, New York, NY, USA, 35–44.
[5] Wasylkowski, A., Zeller, A. Mining temporal specifications from object usage. Autom Softw Eng 18, 263–292 (2011).
[6] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: bug detection with n-gram language models. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16). Association for Computing Machinery, New York, NY, USA, 708–719.
[7] E. Charniak. Statistical Language Learning. In First MIT Press paperback edition. MIT Press, 1996.

[8] S. Baadel, F. Thabtah and J. Lu, "Overlapping clustering: A review," 2016 SAI Computing Conference (SAI), London, UK, 2016, pp. 233-237, doi: 10.1109/SAI.2016.7555988.
[9] Ienco, D., & Bordogna, G. (2016). Fuzzy extensions of the DBScan clustering algorithm. Soft Computing, 22, 1719-1730.
[10] W. Lai, M. Zhou, F. Hu, K. Bian and Q. Song, "A New DBSCAN Parameters Determination Method Based on Improved MVO," in IEEE Access, vol. 7, pp. 104085-104095, 2019.
[11] Luna J M, Fournier-Viger P, Ventura S. Frequent itemset mining: A 25 years review[J]. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 2019, 9(6): e1329.
[12] Zhao Q, Bhowmick S S. Association rule mining: A survey[J]. Nanyang Technological University, Singapore, 2003, 135.
[13] McNicholas P D, Murphy T B, O'Regan M. Standardising the lift of an association rule[J]. Computational Statistics & Data Analysis, 2008, 52(10): 4712-4721.
[14] Fournier-Viger, Philippe, Ted Gueniche, and Vincent S. Tseng. "Using partially-ordered sequential rules to generate more accurate sequence prediction." Advanced Data Mining and Applications: 8th International Conference, ADMA 2012, Nanjing, China, December 15-18, 2012. Proceedings 8. Springer Berlin Heidelberg, 2012.
[15] Philippe Fournier-Viger, Roger Nkambou, and Vincent Shin-Mu Tseng. 2011. RuleGrowth: mining sequential rules common to several sequences by pattern-growth. In Proceedings of the 2011 ACM Symposium on Applied Computing (SAC '11). Association for Computing Machinery, New York, NY, USA, 956–961.
[16] Jian Pei et al., "Mining sequential patterns by pattern-growth: the PrefixSpan approach," in IEEE Transactions on Knowledge and Data Engineering, vol. 16, no. 11, pp. 1424-1440, Nov. 2004, doi: 10.1109/TKDE.2004.77.
[17] eliben. 2010. PyCParser. https://github.com/eli-ben/pycparser.
[18] M. Nayrolles, N. Moha and P. Valtchev, "Improving SOA antipatterns detection in Service Based Systems by mining execution traces," 2013 20th Working Conference on Reverse Engineering (WCRE), Koblenz, Germany, 2013, pp. 321-330, doi: 10.1109/WCRE.2013.6671307.
[19] Bernard Kamsu-Foguem, Fabien Rigal, Félix Mauget,Mining association rules for the quality improvement of the production process,Expert Systems with Applications,Volume 40, Issue 4,2013,Pages 1034-1045,ISSN 0957-4174.
[20] Kim, I., de Weck, O. Adaptive weighted-sum method for bi-objective optimization: Pareto front generation. Struct Multidisc Optim 29, 149–158 (2005).
[21] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang et al., "Codebert: A pre-trained model for programming and natural languages", 2020.
[22] D. Bahdanau et al., "Neural machine translation by jointly learning to align and translate", Proc. Int. Conf. Learn. Representations, 2015.
[23] Zhou Y, Zhan W, Li Z, et al. DRIVE: Dockerfile Rule Mining and Violation Detection[J]. arXiv preprint arXiv:2212.05648, 2022.