Graph-Based Pattern-Oriented, Context-Sensitive Source Code Completion

Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen
Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, Tien N. Nguyen
Electrical and Computer Engineering Department
Iowa State University
{*anhnt,tung,hoan,atamrawi,hungnv,jafar,tien*}*@iastate.edu*

***Abstract*—Code completion helps improve developers' programming productivity. However, the current support for code completion is limited to context-free code templates or a single method call of the variable on focus. Using software libraries for development, developers often repeat API usages for certain tasks. Thus, a code completion tool could make use of API usage patterns. In this paper, we introduce GraPacc, a graph-based, pattern-oriented, context-sensitive code completion approach that is based on a database of such patterns. GraPacc represents and manages the API usage patterns of multiple variables, methods, and control structures via graph-based models. It extracts the context-sensitive features from the code under editing, e.g. the API elements on focus and their relations to other code elements. Those features are used to search and rank the patterns that are most fitted with the current code. When a pattern is selected, the current code will be completed via a novel graph-based code completion algorithm. Empirical evaluation on several real-world systems shows that GraPacc has a high level of accuracy in code completion.**

***Keywords*-pattern-based code completion; API usage pattern**

I. Introduction

Code completion, which is a built-in support in many integrated development environments (IDEs), aims to help improve programming productivity. However, most of the state-of-the-art code completion tools [1], [2] can help in the completion of only a single method call or a single object declaration/initialization. For example, Eclipse provides an alphabetical-order list of available methods for the variable under editing. The user then has to go through a long list of unranked options. For better auto-completion, several approaches [2], [3], [4] have been proposed to rank such a suggested list of method calls. However, the volume of auto-completed code is still limited to a single method call. Aiming to provide more auto-completed code, some IDEs recommend the templates for common program constructs (e.g. `for`, `while`) and popular classes/methods (e.g. `Iterator`). However, they do not consider the context of the code under editing, thus do not predict well users' editing intention.

Using software libraries, developers follow the API specifications to perform their tasks with some intended functionality. Other developers may also repeat those tasks via the same API usages. The correct and repeated ways of using APIs are often called *API usage patterns*. The patterns include the usages of variables, method calls, and control structures, with specific control and data dependencies and the orders among them. Thus, to support better API usages and provide more auto-completed code, a code completion tool can rely on API usage patterns. Moreover, a developer uses the APIs to achieve a different goal in the context of his own program. Thus, such a tool should predict his intention based on the current context of the code under editing (e.g. current API elements) to provide best fitted patterns.

This paper introduces GraPacc, a graph-based, pattern-oriented, context-sensitive code completion method and tool that performs code completion based on a collected database of API usage patterns. Each pattern is represented as a graph-based model [5] that is able to capture the usages of multiple variables in different types, method calls in multiple libraries, control structures, and their data/control dependencies. GraPacc extracts the context-sensitive features from the code under editing, e.g. the API elements being on focus or under modification and their relations to other code elements. The features are then used to search and rank the patterns that are best matched with the current code. When a pattern is chosen by a user, GraPacc will fill in the code based on that pattern with proper replacements of program elements in the current context of the program.

We conducted an empirical evaluation on GraPacc's correctness. The evaluation results on 24 real-world systems show that, GraPacc can achieve up to 95% precision, 92% recall, and 93% f-score in code completion. We found that on average 71% of an API's usage in a project is covered by usage patterns. The key contributions of this paper include:

1. A formulation and algorithms for *graph-based feature extracting*, *searching* and *ranking* of API usage patterns that are best matched with the context of the current code,

2. A novel algorithm for *graph-based code completion*,

3. GraPacc, a graph-based, context-sensitive code completion tool that takes into account the current editing context and makes use of API usage patterns, and

4. A comprehensive empirical evaluation method that shows the correctness and usefulness of GraPacc.

Section II presents motivating examples. Sections III-VI describe our model and algorithms. Our empirical evaluation is presented in Section VII. Related work is discussed in Section VIII. Conclusions appear last.

978-1-4673-1067-3/12/$31.00

```
1  Display display = new Display();
2  Shell shell = new Shell(display);
3  ...
4  Button button = new Button(shell, SWT.PUSH);
5  button.setText("OK");
6  button.setSize(new Point(40,20));
7  button.setLocation(new Point(200,20));
8  ...
9  shell.pack();
10 shell.open();
11 while (!shell.isDisposed()) {
12    if (!display.readAndDispatch())
13       display.sleep();
14 }
15 display.dispose();
```

Figure 1. SWT Usage Example 1

II. Motivating Examples

API Usage Patterns. Figure 1 shows a portion of code using SWT, a graphical user interface (GUI) library, to create a window containing "OK" button. According to SWT documentation, creating a window involves two types of objects. A Display object is responsible for managing the GUI events and related resources between SWT and the operating system. A Shell object represents a GUI window, which is associated with the Display object and acts as a container for other GUI elements such as buttons, text boxes, etc. After a Display object (display) is created, a Shell object (shell) is created to form a top-level window (lines 1-2). It then starts to receive and process the events via display until object shell is disposed (lines 9-14). Finally, object display is disposed.

According to SWT documentation, this usage of those SWT elements (the classes Display, Shell, and their methods) is a correct way to perform the task of creating a top-level GUI window in SWT. Thus, this API usage (lines 1-2 and 9-15) occurs very frequently in Java code that uses SWT library. We call this correct usage an *API usage pattern* (or a pattern for short).

This example shows a common practice in which developers reuse existing software components via libraries and their APIs. Such API elements are intended to be used in some *specific usage procedure* (i.e. an usage pattern) in order to accomplish a specific task. Thus, when using libraries, developers usually follow the corresponding usage patterns to achieve their goals.

Aiming to reduce developers' burden in remembering exact API elements and their usages as well as providing more auto-completed code, a code completion tool could be based on API usage patterns. With the knowledge on usage patterns, it could recommend the appropriate pattern that fits with the current *context* of code under editing. For example, if a developer finishes the first two lines in Figure 1, a pattern-oriented code completion tool should recognize that (s)he is using two SWT variables of types Display and Shell, which belong to the window creation pattern. Then, it could predict that (s)he intends to create a window, and fill in the remaining part of the pattern (lines 9-15). Providing higher volume of code, such a pattern-oriented tool could help to complete code faster than the single-method suggestions.

```
1 Button button = new Button(shell, SWT.PUSH);
2 button.setText ("OK");
3 FormData bData = new FormData();
4 button.setLayoutData (bData);
```

Figure 2. SWT Usage Example 2

Alternative Patterns. Figure 1 (lines 4-7) illustrates an SWT usage pattern for creating a Button object. A Button object can be instantiated with two parameters: one for its SWT window container, and one for the button style. Then, other properties of the button could be set including its textual label (via method setText), its size (via setSize), and its location (via setLocation). SWT also provides another pattern in which the layout (e.g. size and location) of a button is controlled indirectly via a FormData object. In Figure 2, instead of calling setSize and setLocation, one can specify the layout information via a FormData object (line 3), and associate it with the button (line 4).

This example shows that there exist multiple usage patterns for a specific task (called *alternative patterns*). Alternative patterns can share or involve different API elements (classes/methods). Thus, a pattern-oriented code completion tool must be *context-sensitive*, i.e. consider the *context* of the code under editing such as the currently used API elements.

For example, assume that a developer wrote the code to create a Button object and finished lines 1-3 (Figure 2). Due to the existence of a FormData, the tool could predict that (s)he intends to use the pattern of creating a button with a layout object. Thus, the tool should recommend toward the pattern with layout, i.e. ranking it higher than its alternative.

Interleaving Patterns. Figure 1 also illustrates a common case that a developer can use multiple usage patterns whose code is *interleaved* with one another (e.g. creating a window and a button). This implies that a pattern-oriented, context-sensitive code completion tool could use the current focus position as context information to guess the user's editing intention. Then, the tool should switch between those patterns for code completion depending on the current focus cursor. For example, if (s)he is in the middle of editing line 2 (Figure 1), the tool must recognize that the usage pattern for SWT window creation is the most relevant and suggest that pattern (lines 9-15). Assume that during filling the details of that pattern, (s)he switches to create a new Button as in line 4: new Button(...). The tool must recognize the switching and suggest the button creation pattern.

Graph-based Patterns. As seen, some API elements have strict usage orders, while others do not. For example, in the button creation pattern (lines 4-7, Figure 1), the constructor of Button must be called before other methods setText, setSize,

```
1 Display display = new Display();
2 Shell shell = new Shell(display);
3 ...
4 Button button = new Button(shell, SWT.PUSH);
5 FormData formData = new FormData();
6 button._
```

Figure 3. SWT Query Example

and setLocation. However, there is no required order among those setters. The existence of such partial control/data dependencies suggests that, the patterns and the context should be modeled via *graph-based* structures, rather than sequences [6] or sets of method calls [2] as in existing work.

III. Important Concepts

We develop GraPacc, a Graph-based Pattern-oriented, Context-sensitive tool for Code Completion. It takes as an input a database of usage patterns and completes the code under editing based on its context and those patterns.

Definition *1 (***Pattern):** *An API usage pattern is a set of API elements (i.e. classes/variables/method calls) and control structures (i.e. condition/repetition) with specific control and data dependencies. A usage pattern specifies a correct usage of API elements to perform a programming task.*

Figure 1 (lines 1-2, 9-15) shows an instance of SWT window creation pattern. *An instance* is concrete code realizing that pattern. A pattern contains the usage of the classes (via variables), methods (via method calls), and control structures (e.g. while, if), with specific orders and inter-dependencies. A pattern could be a composite one built from multiple sub-patterns. The patterns could be interleaved with each other.

Definition *2 (***Query):** *A query is a code fragment under editing, i.e. a sequence of textual tokens written in a programming language.*

A query is generally incomplete (in term of the task that is intended to achieve) and might not be parsable. Figure 3 illustrates a code fragment as a query. The character _ denotes the editing cursor where a developer invokes the code completion tool during programming.

We developed a graph-based model, called *Groum* [5] to represent object usage patterns. GraPacc uses Groum to represent API usage patterns and queries as follows.

Definition *3 (***Groum):** *A Groum is a graph-based model, in which the nodes represent actions (i.e. method calls), data (i.e. objects/variables), and control points (i.e. branching points of control structures such as if, while, for, etc). The edges represent the control and data dependencies between the nodes. Labels of the nodes are built from the corresponding names of classes/methods/control structures.*

Figure 4 illustrates the usage patterns in Section II as Groum models. For clarity purpose, we label the nodes using the simple names of classes and methods. In the actual implementation, nodes' labels have their fully qualified names.

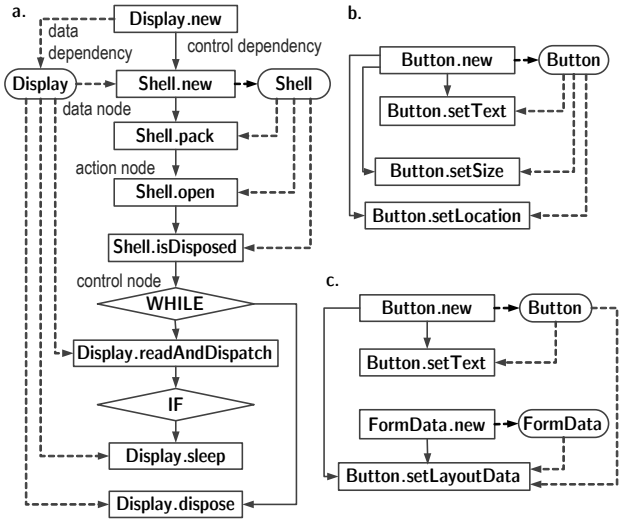

Figure 4. SWT Usage Patterns

As seen, two data nodes labeled Display and Shell represent two variables display and shell. Action nodes such as Display.new, Display.readAndDispatch, and Shell.open represent the method calls. An edge from data node Display to action node Shell.new represents the data dependency, while an edge from Display.new to Shell.new represents their control flow order.

GraPacc matches the patterns to the code under editing via their features extracted from their Groums and associated context-sensitive information. Moreover, because a query might be incomplete or not fully parsable, the corresponding Groum might not contain all necessary information or might not even be available. Thus, GraPacc also extracts the features from the texts of the query. There are two types of features: *graph-based features* and *token-based features*.

Definition *4 (***Feature):** *A graph-based feature is a sequence of the textual labels of the nodes along a path of a Groum. A token-based feature is a lexical token extracted in a query.*

The *size* of a graph-based feature is defined as the number of elements in its corresponding sequence. Thus, in a Groum, a node has a corresponding graph-based feature of size 1, and an edge has a graph-based feature of size 2. Larger features can be built from a path in the Groum. In Figure 4a, there are a size-1 graph-based feature [Shell.new], a size-2 graph-based feature [Shell.new, Shell.pack], a size-3 graph-based feature [Shell.pack, Shell.open, Shell.isDisposed], etc.

In GraPacc, a token-based feature always has its size equal to 1 and is used to represent the usage of a *class*, a *method*, or a *control structure* in the current (incomplete) code. For example, the query `for (Iterator _` is incomplete and can not be parsed into an AST. However, GraPacc still extracts two tokens for and Iterator, and uses them to match this query to the patterns that have the usages with a for loop and an Iterator variable.

To measure the similarity of any two features, GraPacc defines a function sim that compares their textual similarity and the orders of their elements (see Section IV for details).

To compare a query against a pattern via features, GraPacc also takes into account the context information of the query. Such information is modeled via the *context-sensitive weights* associated with the features. That is, context-sensitive weights measure the significance of the features in a query based on the relations of the features to the focus editing position (user-based factor) and based on the structure of the query's Groum (structure-based factor). Based on the similarity of the features and their corresponding context-sensitive weights, GraPacc defines a relevance measure fit between a query and a pattern, in order to rank the candidate patterns to a query. The details of function fit and weights are presented next.

IV. Query Processing and Feature Extraction

GraPacc analyzes the query Q (i.e. the code under editing) and extracts its context-sensitive features and weights in four main steps: 1) tokenizing the input Q to extract lexical tokens, which could be used as token-based features; 2) using Partial Program Analysis (PPA) tool [7] to parse the input code into an AST; 3) building the corresponding Groum from the AST; and 4) extracting the graph-based features from that Groum, collecting the token-based features from the un-parsable tokens (i.e. the tokens without associated AST node), and determining the context-sensitive weights for the extracted features.

1) Tokenizing: GraPacc breaks the code Q within the current method into lexical tokens, records their locations, and computes their distances to the editing cursor. After tokenizing, GraPacc keeps the keywords related to the control structures (e.g. while, if, for, case, etc) and object instantiation (new). Unrelated keywords (e.g. public, class, void, etc) are not used in query formulating but kept for later code completing.

2) Partial Parsing: If the current code under editing is not parsable by Eclipse's Java parser, GraPacc will use the PPA tool [7] to handle the query. The PPA tool, as an Eclipse's plugin, accepts a portion of code and returns an AST with all possible type binding information. However, in some cases, there might exist some unresolved nodes, for example, their types are undeterminable in the query. Thus, they are assigned with an UNKNOWN type.

3) Groum Building: GraPacc constructs the corresponding Groum from the AST provided by PPA in the previous step using the constructing algorithm from our prior work [5]. Due to the incompleteness of the query code, the unresolved nodes in the AST are discarded. They are considered as tokens and used to extract token-based features. The data nodes corresponding to the variables of the data types that are not resolved to fully qualified names are kept with only simple names. Figure 5 shows the Groum built for the query example in Figure 3. As seen, the objects shell, button, bData, and display are resolved to the data nodes labeled with their types Shell, Button, FormData, and Display, respectively. Node Button is denoted as the *focus node*, because the token closest to the editing cursor is button.

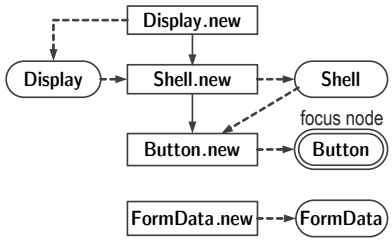

Figure 5. Graph-based Usage Model of Query in Figure 3

4) Feature Extracting and Weighting: In this step, GraPacc extracts the graph-based features from the Groum built for the query, and other features for the retained tokens.

Feature Extracting. GraPacc first maps each node in the Groum built in the previous step back to the tokens built in the Tokenizing step. For example, data node Button in the Groum drawn in Figure 5 is mapped to three tokens of the query listed in Figure 3: Button (line 4), button (line 4), and button (line 6). The first token denotes the type annotation of the variable button corresponding to that data node, and the two other tokens are the two references of that variable. After the mapping, any token that does not correspond to any node in the Groum is selected as a token-based feature. Next, different features are extracted from various paths in the Groum. Since there might be a large number of paths, only the paths with limited sizes ($L \leq 3$) are considered. This limit were determined experimentally in our prior work to achieve high accuracy in Groum matching [5]. Moreover, using large-size features reduces performance significantly because the number of features increases exponentially to the maximum size of features. From now on, we use the graph-based feature and its corresponding path interchangeably.

Feature Weighting. When a feature (graph-based/token-based) is extracted, its weight representing its context-sensitive significance is also computed as follows:

$$w(q) = (w_s(q) + w_c(q)) \times w_f(q) \qquad (1)$$

a. $w_s(q)$ indicates the structure-based factor of feature q via its size (from 1 to 3): $w_s(q) = 1 + size(q)$. That is, a longer feature represents more information, and is assigned with higher weight. The rationale is that a long feature allows GraPacc to capture stricter dependencies among several nodes in the path for that feature. The addition of 1 aims to reduce the relative difference between features of different sizes, e.g. if $minsize$=1, $maxsize$=3, then (3+1)/(1+1) $<$ 3/1, thus, making the effect of the size feature on the final weight in formula (1) smoother.

b. $w_c(q)$ models the structure-based factor of feature q via the centrality of the corresponding nodes in the Groum. The rationale is that if a node has high centrality in a Groum, it plays an important role and can be better used for matching. For example, feature Button.new is considered to be more important than FormData.new in the query of Figure 5 because the corresponding node for Button.new has more dependencies to other nodes. Thus, if feature q has size s and the nodes of the path corresponding to q have n neighbors, $w_c(q) = n/s$.

c. $w_f(q)$ models the user-based factor of q via its relation to the current editing position, i.e. the focus node. For example, Button is the focus node. Thus, feature Button.new is considered to be more important than Shell.new. $w_f(q)$ is computed based on the distance d between the focus node and the path from which feature q is extracted: $w_f(q) = 1/(d+1)$. d is computed as the length of the shortest path from the focus node to a node in that path. Thus, if that path contains the focus node, d is 0, and $w_f(q)$ is maximized. If the path contains only the neighbors of the focus node, d is 1, and $w_f(q)$ is 0.5.

If q is a token-based feature, its size is 1, thus, its size-based weight w_s is the same as the weight of a graph-based feature of size 1. Its centrality-based weight w_c is 0, because no structural information is available. Its focus-based weight w_f is $1/(d+1)$, with d being its distance to the token closest to the focus editing point in the Groum.

In the formula (1), $w_s(q)$ and $w_c(q)$ are added together while $w_f(q)$ is multiplied since $w_s(q)$ and $w_c(q)$ represent structure-based factors (feature's size and centrality) and $w_f(q)$ is for user-based factor (distance to the focus point). They are context-sensitive information in different spaces.

V. Pattern Managing, Searching and Ranking

Pattern Management. The patterns can be automatically imported from the mining results of the pattern mining tool, GrouMiner [5], or be manually provided by the users. Each pattern is stored as a Groum along with a textual template code fragment [5]. A parameter $Pr(P)$ is stored to represent the popularity of pattern P. For the patterns mined from codebase, GraPacc uses their occurrence frequencies in the codebase for $Pr(P)$. For user-provided patterns, the user can either specify this parameter or a default value is assigned.

To support efficient searching of patterns based on features, GraPacc uses an inverse indexing mechanism. It extracts the graph-based features from a pattern, and for each feature p, it stores the list $L(p)$ of patterns from which feature p could be extracted. For each extracted feature p, GraPacc uses a weight $s(p, P)$ to represent its significance in each pattern P containing the feature p. The weight $s(p, P)$ is computed based on the Tf-Idf weighting scheme [8]:

$$s(p, P) = N_{p,P}/N_P.(logN - logN_p)$$

$N_{p,P}$ is the number of occurrences of feature p in P,

N_P is the total number of features in P,

N_p is the number of patterns containing feature p, and

N is the total number of patterns in the pattern database.

The inverse indexing list of patterns for each feature is sorted according to those weights.

Searching and Ranking. Another crucial task is to search and rank a list of relevant patterns for the code under editing (i.e. a query). The core step is to compute the relevance degrees of the candidate patterns to that query based on the features and context-sensitive factors/weights computed from the query. However, there are two following challenges:

a. Due to the incompleteness of the query, there might be some extracted features that do not exist in the pattern database (e.g. the features for the nodes whose types are *unresolvable* to fully qualified names). Thus, the features in the pattern database (called *pattern features*) might not exactly match to the features in the query (called *query features*).

b. The number of patterns in a database is often large, it is inefficient to compute the relevance degrees for all patterns.

For issue a, GraPacc uses the similarity function sim, which will be explained next, to find the features existing in the pattern database that are best-matched to the query features. If p is a pattern feature, q is a query feature, and $sim(p, q) \geq \delta$, with δ being a pre-chosen threshold, then p is added to the set F of the mapped features for q. GraPacc uses this set to solve issue b. For each pattern feature $p \in F$, the top-n ranked patterns in its ranked inverse indexing list $L(p)$ are added to the list of candidate patterns C for the relevance computation for q. After this step, GraPacc computes the relevance measure function $fit(P, Q)$ of each candidate pattern $P \in C$ to the query Q, ranks them based on those relevance degrees, and returns the ranked list of patterns. Let us describe the functions sim and fit.

1) Feature Similarity sim**.** Function sim computes the similarity between two features. Both graph-based features and token-based features could be considered as a sequence of labels/names, thus their similarity is computed mainly based on the names of those labels. GraPacc defines the similarity only for two features of the *same size*. The similarity of two features p, q of size k is computed as:

$$sim(p, q) = \prod_{i=1}^{k} nsim(p_i, q_i) \quad (2)$$

in which $nsim$ is the name-based similarity measure, and p_i and q_i are the i-th element of p and q, respectively. The similarity degree of features with different sizes is zero.

In GraPacc, a standard label p_i has the following form $X.Y.Z$, in which X is the qualified name of the package, Y and Z are the simple names of the class/method, respectively. X, Y, or Z might be empty. For example, for a data node, Z is empty. Sometimes, X is empty since the package name is unresolvable in the query. Thus, for two

labels $X.Y.Z$ and $X'.Y'.Z'$, its name-based similarity $nsim$ is defined as

$$\frac{\alpha \times wsim(X, X') + \beta \times wsim(Y, Y') + \gamma \times wsim(Z, Z')}{\alpha + \beta + \gamma} \quad (3)$$

in which α, β, and γ are weighting parameters, and $wsim$ is a word-based similarity value. If in a label, two corresponding parts are missing, the corresponding term in formula (3) is discarded. For example, if neither labels have the X parts, the first term and its weight parameter α are discarded.

To compute the word-based similarity $wsim$ of two strings X and X', GraPacc first breaks them into single words using Camel convention. For example, StringBuffer is broken into two words String and Buffer. Then, the similarity of two labels, viewed as two sequences of words $L(x)$ and $L(y)$, is defined as L_o/L_m, in which L_o is the length of their longest common subsequence, and L_m is the average length of two sequences. This scheme enables GraPacc to support incompletely-typed and non-exact matched entity names.

GraPacc considers a token-based feature T (size 1) as comparable to a graph-based feature of size 1 (with some label $X.Y.Z$), because a token could be the name of a variable or a method in the query and should be comparable to the label of a Groum's node of a pattern. In this case, $nsim$ is defined as

$$max(wsim(T, X), wsim(T, Y), wsim(T, Z)) \quad (4)$$

The max function is used since a token in the current code could be the name of either a package, class, or method.

2) Pattern Matching. GraPacc models two patterns P and Q as two sets of features, each feature has its own significance weight, and each pair of features has the similarity measured by function sim. Thus, the relevance measurement between P and Q is based on the *weighted maximum bi-partite matching*, i.e. matching each feature of P to a feature of Q in order to maximize the total similarity and significance between all matched pairs of features in P and Q. The relevance degree between a pair of features $p \in P, q \in Q$ is computed as:

$$relevance(p, q) = s(p, P) \times sim(p, q) \times w(q) \quad (5)$$

- $s(p, P)$: the significance of feature p in pattern P according to the Tf-Idf scheme,
- $w(q)$: the context-sensitive significance of q in query Q,
- $sim(p, q)$: the similarity of two features.

The maximal weighted match for P and Q is a map M for each feature p of P to an unique feature q of Q such that the total weight of matched pairs

$$S_M(P, Q) = \sum_{p \in P, q = M(p)} relevance(p, q) \quad (6)$$

is maximal among all possible maps. Because GraPacc also considers the popularity $Pr(P)$ of a candidate pattern, the relevance degree of the pattern P to the query Q is computed as follows:

$$fit(P, Q) = S_M(P, Q) \times Pr(P) \quad (7)$$

VI. Pattern-Oriented Code Completion

If the user chooses a pattern P in the recommended list, GraPacc will complete the code in the query Q according to pattern P. Generally, to do that, GraPacc first matches the code in P and Q to find the code in P that has not appeared in Q. Then, it fills such code into Q in accordance with the context in Q, i.e. at the appropriate locations in Q and with the proper names.

Let us first explain the general idea via an example. Let us revisit the query example in Figure 5 (the corresponding code is in Figure 3) and assume that a user selects pattern c) in Figure 4 (the corresponding code is in Figure 2). GraPacc first determines that the two Button.new nodes, the two FormData.new nodes, the two Button nodes, and the two FormData nodes in the two Groums are respectively matched. That is, two object initializations and the assignment to the variables for Button and FormData already existed in the query. Compared with pattern P, the nodes that have not used include Button.setText and Button.setLayoutData. Thus, GraPacc uses the code corresponding to those nodes to fill in Q.

The code completing task is done via creating the corresponding sub-trees in the AST of Q at the appropriate positions and with the proper names for the fields and variables. For example, to fill in Button.setLayoutData, it first needs to create that method call and find its position in the AST of Q (not shown). In this case, the position is next to the variable node button in the AST of Q. Since in the pattern, Button.setLayoutData has a parameter of type FormData (Figure 4c), GraPacc must fill in that parameter with a proper name. From pattern P, that parameter must be from the FormData node (Figure 4c), which is matched to FormData in Q (Figure 5). It in turn corresponds to the variable formData in Q (Figure 3). Thus, GraPacc chooses the name formData and fills in line 6 of Figure 3. Similar process is used for Button.setText, which is added between lines 4-5 of Figure 3. Therefore, the final result is:

```
1 Button button = new Button(shell, SWT.PUSH);
2 button.setText(_);
3 FormData formData = new FormData();
4 button.setLayoutData(formData);
```

Let us describe the algorithm in details.

A. Matching Groum Nodes in Pattern and Query

GraPacc performs code matching on Q and P on their Groums, i.e. for each node v in P, it determines the best matched node u in Q (Figure 6). To do so, it retrieves two sets of features $F(u)$ and $F(v)$ corresponding to the paths through u and v, respectively. It then runs a weighted

```
1 function GroumNodeMatching(G_Q, G_P)
2 for each node u in G_Q
3   for each node v in G_P
4     // finding best matching between two sets of features
5     BipartiteMatching(F(u), F(v), relevance(p, q)) with p ∈ F(u),
          q ∈ F(v)
6     match(u, v) = max(∑{relevance(p, q)}) //matching level for (u, v)
7 // finding the sets of best−matched nodes in P and Q
8 BipartiteMatching(G_Q, G_P, match(u, v))
9 Return the mapping M for the nodes in G_Q and G_P
```

Figure 6. Groum Node Matching between Pattern P and Query Q

bipartite matching algorithm with the weights being measured via $relevance$ function (line 5). The matching degree between u and v is measured by the sum of the relevance degrees corresponding to the best matching (line 7). After computing all matching degrees for all u and v, GraPacc performs bipartite matching to find maximal aligned sets of nodes in Q and P (line 9). Then, it returns the mapping M, i.e. $M(v) = u$ means that $v \in P$ is matched to $u \in Q$, and $M(v) = null$ if v is not matched to any node in Q. For example, while matching the Groums for Q in Figure 5 and for P in Figure 4c, it determines that Button.new, FormData.new, Button, and FormData have matches. Button.setText and Button.setLayoutData are unmatched nodes.

B. Completing the Query Code

After having the mapping, GraPacc performs code completing (Figure 7). It traverses the un-matched nodes in the Groum of pattern P in a breadth-first order and for such a node, it finds the corresponding AST's subtree at that node in the AST of pattern P via the stored template code of P. Then, it clones that sub-tree (line 4) and updates the name attributes of the nodes of that sub-tree in accordance with the code in Q (line 5). After that, it finds the proper position for that sub-tree in the AST of Q (line 6) and attaches it to the AST via Eclipse's AST editing support (line 7).

1) Finding Appropriate Names for Variables before Filling-in (updateName): Since variables in P and Q generally are named differently, to be able to fill in a variable in P into Q, GraPacc needs to update its name accordingly. For example, although two data nodes FormData in Figure 4c and Figure 5 are matched, the corresponding variables in ASTs are bData and formData. To find such proper name, GraPacc uses the mapping M: if node $v \in P$ is matched to $u \in Q$, then the relevant name for the variable involving v will be u's name; if v is unmatched, but is the reference/declaration of a variable corresponding to a matched node $v' \in P$, the relevant name for the variable involving v will be v''s name. Otherwise, the relevant name for v will be kept the same as in P. However, to avoid accidental duplicate names in P with those in Q as the code is filled in at the next step, for all nodes that are not matched and not renamed, if they have the same names with any nodes in Q, they are renamed with new indexes being added.

```
1 function CodeCompletion( M, P, Q)
2   //cloning AST nodes of the unmatched nodes from P to Q
3   for each node v such that M(v) = null:
4     T = clone(AST_P, v)
5     updateName(T, M)
6     pos = findPosition (AST_Q,T)
7     updateQueryCode(T, AST_Q, pos)
```

Figure 7. Code Completion from Pattern P to Query Q

Table I
TRAINING DATA FOR JAVA UTILITY PATTERNS

Project	Files	Methods using Java Util	Mined Patterns
EclipseME	137	619	28
AspectJ	1,053	5,859	155
Codehaggis	20	52	4
Unitmetrics	34	103	10

2) Finding the Position for an Unmatched Node v in P within the AST of Q (findPosition): Its position is determined via the relative position of v with respect to the matched nodes in its neighbors in P. For example, to find the location to fill Button.setText into Q, GraPacc determines that in P, that node follows Button.new. According to the sequential order in the code of P (Figure 2), it comes before FormData. With the mapping for those nodes, its location is determined as between two AST nodes corresponding to line 4 and line 5 of Figure 3. The following neighboring relations of v in a pattern are used to determine the relative positions:

- v is the initialization of a variable declaration,
- v is a parameter of a method invocation,
- v is in a conditional expression or the body of an if node,
- v is a control node/ method call having the matched nodes.
- v is a node having a sequential order with matched nodes.

If GraPacc cannot find the relative position for v (e.g. no matched node as a pivot), the current focus point is used.

Note that GraPacc's code completion can be invoked on demand at any point in the currently edited code. It can search for a pattern that appears non-contiguously since it captures control/data dependencies among the elements in an API usage backward and forward from the invoking point. Thus, it can support both programming styles: writing line-by-line, and creating code skeleton and then filling in.

VII. EMPIRICAL EVALUATION

This section presents our experimental studies to evaluate GraPacc's accuracy in code completion. GraPacc is realized as an Eclipse plug-in. All experiments were carried out on a machine with CPU AMD Phenom II 3.0 GHz, 8GB RAM.

A. Experiment Setting

Java SDK Utility (java.util, java.io) [9] was chosen since it contains a rich set of usages and many open-source systems have used its APIs. We collected a total of 28 open-source Java projects using Java Utility library. We then used our pattern mining tool, GrouMiner [5], to collect API patterns

```
1 Scanner scanner = new Scanner(new File ("C:/sample.dat"));
2 ArrayList<String> list = new ArrayList<String>();
3 while(scanner.hasNext()) {
4     list .add(scanner.next());
5 }
6 StringBuffer strBuf = new StringBuffer();
7 Iterator itr = list . iterator ();
8 while ( itr .hasNext())
9 {
10    String str = itr .next() + ":";
11    strBuf.append (str);
12 }
13 System.out.println(strBuf. toString ());
```

Figure 8. An Example of a Test Method

of Java Utility from a set of 4 Java projects, which were used as the tool's knowledge (Table I). Other 24 projects were used for evaluation (Table II). Eventually, we had 197 patterns in our database with 1,288 features.

We built an automatic evaluation tool and for each subject project, we first used it to collect all methods using Java Utility. For such a method, we simulated a real programming situation. We assumed that a developer partially finished his/her coding in that method and requested the help from GraPacc. Thus, we divided the code of the method under testing (called a *test method*) into two parts: the first part was used as a query, and the second for evaluation.

We followed a similar automatic evaluation process for a code completion tool as in Bruch *et al.* [2]. Let us explain the procedure of handling a test method via an example in Figure 8. Our evaluation tool first collected from the test method all occurrences of the API elements including method calls, object creation, data variables, and control structures that are related to Java Utility. It sorted them in the order of their occurrences in the test method. The one at the middle position of that sorted list was chosen as the cut point (focus point). The first part of the test method from its beginning to the cut point was used as a query for evaluation. The rationale for this way of selecting a focus point at the middle point is to avoid the cases in which no Java Utility API element appears in the first part or none of them is left in the second part of the test method. For Figure 8, the query is as follows:

```
...
StringBuffer strBuf = new StringBuffer();
Iterator itr =_
```

B. Evaluation Metrics

For each given query, GraPacc was invoked and it returned a ranked list of patterns. Assume that a pattern was selected, and GraPacc would complete the code. Let us use O and R to denote the original and the resulting code (from GraPacc) in the second half of the test method, respectively. As explained, there might be no specific order between two API elements. If we compared directly R to O based on their texts, the evaluation would be imprecise since a correct result from GraPacc might not match exactly the writing order of API elements in O. Moreover, the goal was to evaluate how well GraPacc completed for Java Utility elements (rather than other elements). Thus, we compared the Groum of the resulting code R with that of the original code O.

Let us call their respective Groums G_R and G_O. If a node in G_R matches with a node in G_O, we count it as an *correctly* suggested node. If a node in G_R does not occur in G_O, we count it as an *incorrect* node n (because a user would need to delete the corresponding code from the recommended code). If a node in G_O does not occur in G_R, we consider this as a *missing* node m (i.e. the user would need to manually add the corresponding code after code completion). Note that, the original method O might use API elements that do not belong to Java Utility, in which GraPacc has no knowledge. Thus, we counted only the missing nodes in G_O relevant to that library.

Accuracy is measured via *precision*, *recall*, and *f-score*. Precision is defined as the ratio of the number of correctly recommended nodes over the total number of all recommended nodes. Recall is the ratio of the number of correctly recommended nodes over the total number of completion-needed nodes. We also computed f-score, a harmonic average of precision and recall: *f-score = 2 / (1/ precision + 1/recall)*. Higher f-score means better accuracy.

C. Experiment Procedure

Our evaluation tool ran GraPacc on each test method and a ranked list of patterns was returned. To simulate a real coding situation in which a user would choose the desired pattern (i.e. the most similar one), our evaluation tool selected the pattern with the highest *f-score* in the *top-5 list* of the recommended patterns returned by GraPacc.

A method under test m might contain multiple Java Utility API patterns. Thus, in practice, a user might need to invoke GraPacc multiple times to get sufficient recommendations to complete the second half of m. To simulate that, our evaluation tool iteratively invoked GraPacc at multiple focus points in the second half of m. At each iteration, the tool selected an additional focus point, invoked GraPacc and picked the pattern with highest f-score in the top-5 patterns, and counted the numbers of (in)correct/missing nodes. The process continued until all API elements in the second half were completed or no new API elements/nodes can be correctly added (i.e. *all* added API elements are incorrect). This second condition simulates the case where the user does not find the correct API elements returned by GraPacc and continues coding. In each iteration, for the process to continue, at least one of API elements must be filled. Thus, the maximum number of iterations is equal to the number of API elements in the second part of m.

The selection mechanism for the additional focus points with multiple iterations is based on the variables that existed

Table II
CODE COMPLETION ACCURACY RESULT

System	Methods	Patterns	Variables	Calls	Controls	Correct	Incorrect	Missing	Precision	Recall	F-score
anyedittools	81	95	151	251	74	200	22	58	90.1%	77.5%	83.3%
apache-axiom	598	689	801	1,386	415	1,084	269	509	80.1%	68.0%	73.6%
apache-ivy	1,400	1,923	2,121	4,291	1,620	4,480	580	1,482	88.5%	75.1%	81.3%
apache-roller	1,443	1,738	1,879	3,378	1,147	3,205	536	1,501	85.7%	68.1%	75.9%
Aribaweb	1,866	2,344	4,000	7,057	2,173	5,538	1,340	2,967	80.5%	65.1%	72.0%
cayene	4,476	4,653	5,305	8,072	2,598	6,391	1,560	3,537	80.4%	64.4%	71.5%
cvsgrapher	39	55	57	99	38	95	8	32	92.2%	74.8%	82.6%
dom4j-1.6.1	565	660	764	1,324	415	1,274	107	375	92.3%	77.3%	84.1%
dvsl	46	53	56	67	28	69	4	19	94.5%	78.4%	85.7%
geronimo	92	114	273	398	142	356	88	128	80.2%	73.6%	76.7%
jibx	843	949	1,046	1,675	514	1,412	299	569	82.5%	71.3%	76.5%
Jlibrary	474	612	676	1,253	464	1,385	170	384	89.1%	78.3%	83.3%
jnormalform	194	450	582	1,178	348	1,184	156	254	88.3%	82.3%	85.2%
OPENWFE	1,331	1,687	1,957	4,052	1,256	3,993	598	1,139	87.0%	77.8%	82.1%
PetriEditor	37	50	53	106	49	137	10	12	93.2%	91.9%	92.6%
quack	36	46	67	81	32	64	13	37	83.2%	63.4%	72.0%
RONEditor	366	436	446	838	350	878	144	320	85.9%	73.3%	79.1%
schemaeditor	149	209	262	574	211	606	32	105	95.0%	85.2%	89.8%
sdiff	506	673	943	2,609	1,123	2,405	412	1,131	85.4%	68.0%	75.7%
syper	112	167	191	419	212	375	90	187	80.1%	66.7%	72.9%
varia	158	256	436	949	274	854	128	298	87.0%	74.1%	80.0%
VOCL	189	214	461	733	266	583	66	183	89.8%	76.1%	82.4%
xaware	161	212	222	491	274	498	93	232	84.3%	68.2%	75.4%
xmlrpc	26	28	29	55	35	56	9	33	86.1%	62.9%	72.7%
	15,188	18,313	22,778	41,336	13,990	37,122	6,734	15,492	84.6%	71.0%	77.0%

in the query O and the newly added variables via code completion. The evaluation tool maintains a priority queue D of variables. For the first cut point, this queue D was initialized with all variables in the first half of the test method. The variables with shorter distances to that focus point were placed in the front of D. If a variable appears multiple times, the distance of only its last occurrence is measured to the current focus point. Thus, the list D contains a variable at most once. For example, for Figure 8, initially, D = [itr, strBuf, scanner, list]. GraPacc completed the code at the first iteration as follows:

```
...
StringBuffer strBuf = new StringBuffer();
Iterator itr = list.iterator();
while (itr.hasNext()){
    itr.next();
}
```

To select a new focus point, the evaluation tool considered all variables of any types in the newly added code recommended by GraPacc. It first added those variables in the front of the queue D, based on their distance to the current focus point. If a variable exists in the queue, it will be moved to the front. Finally, the variable that was just processed will be put at the tail of the queue. For example, the queue D was updated as follows: 1) list was moved to the front because it was the only added element, and 2) itr was placed at the tail of D. Thus, D= [list, strBuf, scanner, itr]. The variable at the front of D was then selected to be processed next, i.e. the variable list. The last occurrence of that variable in the new code after completion at this iteration was chosen to be the *next focus point* because its prior occurrences might not provide as much context to expand a new pattern. In the example, the next focus point was at Iterator itr = list.iterator();_.

This scheme of selecting a new focus point simulates the real situation in which a user would focus on the variable that was most recently completed by GraPacc. This procedure is applied to each test method. The numbers of (in)-correct/missing elements are accumulated for all test methods and iterations. Precision, recall, and f-score are computed from the accumulated numbers for entire subject system.

D. Accuracy Result

We ran our evaluation tool with the above procedure. The parameters are chosen as follows: γ=0.6, β=α=0.2, δ=0.9. They are not representative and were chosen after fine tuning for this experiment. Column Methods in Table II shows the number of test methods. Columns Patterns, Variables, Calls, and Controls show the number of the recommended patterns and the numbers of involved variables, method calls, and control nodes in those patterns, respectively. Columns Correct, Incorrect, and Missing display the numbers of (in)correctly recommended and missing API elements. As seen, GraPacc suggested 18,313 API patterns with 22,778 variables, 41,336 calls, and 13,990 control nodes. At each iteration, GraPacc filled in one pattern. In total, it filled in 18,313 patterns for 15,188 methods (Table II). Thus, it took on average 1.2 iterations to converge. It achieves very high accuracy, with up to 95% precision, 92% recall, 93% f-score. The accumulated result shows that precision, recall, and f-score values are 84.6%, 71%, and 77%, respectively. Interestingly, the average recall of 71% suggests that about 71% of an API's usage in a project is covered by API usage patterns.

We also analyzed the incorrect and missing cases and found a few sources of inaccuracy. First, a usage scenario requires an extra API call. This affects GraPacc's accuracy,

however, in practice, users can easily customize usage patterns. The second cause is due to the missing patterns in our evaluation database. The third cause is when an API usage spans two methods and GraPacc's suggestion is redundant.

Time Efficiency. In this experiment, we used GrouMiner [5] to mine the patterns from all 28 subject systems to collect 977 usage patterns in 7 libraries (6,378 API elements, 4,905 distinct features). We ran GraPacc on the same set of 15,188 test methods (Table II). The time for each query with handling, searching, and ranking the candidate patterns, and code filling is about 0.7s. Thus, it is very time efficient.

Threats to Validity. We used a simulation for users' editing actions, rather than true editing. The focus point selection might not reflect well users' editing. Another threat is the insufficient patterns mined from GrouMiner.

VIII. Related Work

Code Completion. Bruch *et al.* [2] propose three code completion algorithms to suggest the method call for a single variable under editing based on code examples in a database. The first one, FreqCCS, suggests the method that is most frequently used in the database. The second one, ArCCS, mines the associate rules $A \rightarrow B$ in which if method A is used, method B is often called and will be suggested.

In contrast to mining a single, most frequently used method call in FreqCCS and the most frequent pair of method calls in ArCCS, GraPacc suggests the usage patterns (i.e. most frequently used graph-based API usages), which contain all involved method calls, variables, and control structures of the usages. Thus, GraPacc represents better the current *context*. Such context is important in code completion (Section II). The features in FreqCCS and ArCCS correspond to individual nodes (for method calls) and individual edges (for pairs of calls) in our Groum. Importantly, GraPacc can handle *multiple variables* in one or *multiple types*, while they focus only on completing the method call for the *single variable* under editing.

The third algorithm, BMN (best-matching neighbors), adapts k-nearest-neighbor algorithm to recommend for a variable v. BMN encodes the current context and the examples in the database as binary feature-occurrence vectors [2]. The features for a context are the *un-ordered set* of method calls of v in the currently edited code and the names of the methods that use v. The set of vectors of examples with the same smallest Hamming distance to the query vector is called the BMN set. Then, BMN ranks the methods based on their frequencies in the examples in the BMN set.

In comparison, GraPacc has several key advances over BMN. First, GraPacc captures richer contextual information of the code under editing, with all *ordered* method calls, *multiple variables*, and control structures in API usages, while BMN represents a context by an *un-ordered set* of method calls of a *single variable*. Second, with the use of API patterns (i.e. correct usages) as a guidance for code completion, GraPacc can make better *context-sensitive* method call completion when there exist *alternative patterns* (Section II). Importantly, it can handle *multiple variables* in *different types* in a usage. Finally, with API usage patterns, GraPacc recommends *more code elements*.

Hill and Rideout [4]'s code completion approach relies on code clones. It matches the fragment under editing with small similar-structure code clones, and then performs transformations for code completion. GraPacc leverages code similarity at the *API-usage* level. Robbes and Lanza [3] propose 6 strategies to improve code completion using recent histories of modified/inserted code during an editing session. GraPacc has an advance in supporting code completion for multiple variables in different types, while their approach focuses on a single method call. Eclipse [1] and other IDEs [10], [11] complete for the call of a variable. Eclipse supports template-based completion for common constructs/APIs (for/while, Iterator) without considering the context.

Example Code Search. MAPO [6] mines and indexes API usage patterns and recommends the associated *code examples*. It does not support auto-completion. Its pattern is sequential rules of method calls. It does not progressively update resulting patterns as context changes. Strathcona [12] extracts the *structural context* of the code under editing and finds its relevant examples. It does not aim for code completion. Structural context includes inheritance relationships, overridden methods, and caller/callee methods of current code. Mylyn [13], a code recommender, learns from a developer's personal usage history and suggests related methods. Personal usage history and structural context could provide the useful guide for GraPacc.

Code searching techniques based on program analysis include Prospector [14], XSnippet [15], PARSEWeb [16], Reiss [17]'s. Other approaches use information retrieval [18], [19], [20], [21]. Static analysis is used to extract API patterns into finite state machine [22], pairs of calls [23], [24], [25], partial orders of calls [26]. Other pattern mining approaches include [27], [28], [29], [30], [31], [32], [33], [34], [35].

IX. Conclusions

We introduce GraPacc, a code completion tool that helps complete the code under editing based on a database of usage patterns. It extracts the context features from the current code and uses them to search and rank the patterns that are best-fitted. As a pattern is chosen, it fills in the code with proper code elements. Empirical evaluation results show that GraPacc can achieve high accuracy in code completion up to 95% precision, 92% recall, and 93% f-score. GraPacc's video demo and tool information can be found at [36].

Acknowledgment

This project is funded by US National Science Foundation (NSF) CCF-1018600 grant. It was also funded in part by Vietnam Education Foundation for the first and fifth authors.

REFERENCES

[1] "Eclipse," www.eclipse.org.

[2] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *ESEC/FSE '09*. ACM, 2009, pp. 213–222.

[3] R. Robbes and M. Lanza, "How program history can improve code completion," in *ASE '08*. IEEE CS, 2008, pp. 317–326.

[4] R. Hill and J. Rideout, "Automatic method completion," in *ASE '04*. IEEE CS, 2004, pp. 228–235.

[5] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based Mining of Multiple Object Usage Patterns," in *ESEC/FSE '09*. ACM Press, 2009.

[6] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *ECOOP 2009*. Springer-Verlag, 2009, pp. 318–343.

[7] B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs," in *OOPSLA '08*. ACM, 2008, pp. 313–328.

[8] G. Salton and C. Yang, "On the specification of term values in automatic indexing," *Journal of Documentation*, vol. 29, no. 4, pp. 351–372, 1973.

[9] "Java sun," java.sun.com.

[10] "Intellisense," http://blogs.msdn.com/b/vcblog/archive/tags/intellisense/.

[11] "Informer," http://javascript.software.informer.com/download-javascript-code-completion-tool-for-eclipse-plugin/.

[12] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *ICSE '05*. ACM, 2005, pp. 117–125.

[13] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *SIGSOFT '06/FSE-14*. ACM, 2006, pp. 1–11.

[14] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the api jungle," in *PLDI '05*. ACM, 2005, pp. 48–61.

[15] N. Sahavechaphan and K. Claypool, "XSnippet: mining For sample code," in *OOPSLA '06*. ACM, 2006, pp. 413–430.

[16] S. Thummalapenta and T. Xie, "Parseweb: a programmer assistant for reusing open source code on the web," in *ASE '07*. ACM, 2007, pp. 204–213.

[17] S. P. Reiss, "Semantics-based code search," in *ICSE '09*. IEEE CS, 2009, pp. 243–253.

[18] "Koders," www.koders.com.

[19] "Google Code Search," www.google.com/codesearch.

[20] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *ICSE '10*. ACM, 2010, pp. 475–484.

[21] Y. Ye, G. Fischer, and B. Reeves, "Integrating active information delivery and reuse repository systems," in *SIGSOFT '00/FSE-8*. ACM, 2000, pp. 60–68.

[22] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *ESEC-FSE '07*. ACM, 2007, pp. 35–44.

[23] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: a general approach to inferring errors in systems code," in *SOSP '01*. ACM, 2001, pp. 57–72.

[24] B. Livshits and T. Zimmermann, "Dynamine: finding common error patterns by mining software revision histories," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 296–305, 2005.

[25] C. C. Williams and J. K. Hollingsworth, "Automatic mining of source code repositories to improve bug finding techniques," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 466–480, 2005.

[26] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining api patterns as partial orders from source code: from usage scenarios to specifications," in *ESEC-FSE '07*. ACM, 2007, pp. 25–34.

[27] M. Gabel and Z. Su, "Javert: fully automatic mining of general temporal properties from dynamic traces," in *SIGSOFT '08/FSE-16*. ACM, 2008, pp. 339–349.

[28] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: mining temporal api rules from imperfect traces," in *ICSE '06*. ACM, 2006, pp. 282–291.

[29] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *POPL '02*. ACM, 2002, pp. 4–16.

[30] S. Shoham, E. Yahav, S. Fink, and M. Pistoia, "Static specification mining using automata-based abstractions," in *ISSTA '07*. ACM, 2007, pp. 174–184.

[31] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Path-sensitive inference of function precedence protocols," in *ICSE '07*. IEEE CS, 2007, pp. 240–250.

[32] D. Lo and S. Maoz, "Mining scenario-based triggers and effects," in *ASE '08*. IEEE CS, 2008, pp. 109–118.

[33] R. Alur, P. Černý, P. Madhusudan, and W. Nam, "Synthesis of interface specifications for java classes," in *POPL '05*. ACM, 2005, pp. 98–109.

[34] T. A. Henzinger, R. Jhala, and R. Majumdar, "Permissive interfaces," in *ESEC'05/FSE-13*. ACM, 2005, pp. 31–40.

[35] C. Liu, E. Ye, and D. J. Richardson, "Software library usage pattern extraction using a software model checker," in *ASE '06*. IEEE CS, 2006, pp. 301–304.

[36] http://home.engineering.iastate.edu/%7Eanhnt/Research/GraPacc/.