Locating Semantically Similar Code Elements via Searching the System Dependence Graph

Xiaoyin Wang[1], David Lo[2], Jiefeng Cheng[3], Lu Zhang[1], Hong Mei[1], Jeffrey Xu Yu[3]
[1]Key Laboratory of High Confidence Software Technologies (Peking University),
Ministry of Education, Beijing, 100871, China
[2]School of Information Systems, Singapore Management University
[3]The Chinese University of Hong Kong, China
{wangxy06,zhanglu,meih}@sei.pku.edu.cn, davidlo@smu.edu.sg, {jfcheng,yu}@se.cuhk.edu.hk

ABSTRACT

In software maintenance and evolution, it is common that developers want to apply a change to a number of similar places. Due to the size and complexity of the code base, it is challenging for developers to locate all the places that need the change. Therefore, techniques that can help developers achieve this purpose should be necessary. In this paper, we propose a technique that enables developers to search for semantically similar code elements satisfying user-defined dependence constraints. Our approach allows developers to make queries involving dependence relationships and textual conditions on the system dependence graph of the program. We carried out an empirical evaluation on four searching tasks in the development history of two real-world projects. The results of our evaluation demonstrate that, compared with code-clone detection and text search, our approach is able to effectively reduce false positives without losing any required code elements.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement

General Terms

Management, Reliability

Keywords

System Dependence Graph, Code Search, Graph Indexing

1. INTRODUCTION

In software development, developers often need to apply one change to a number of similar places in the code. Such a situation occurs typically when developers want to make a programming style change or want to change the environment (e.g., the database system, the operation system, and the user interface) of the software. For example, when a developer wants to extract a common code pattern to form a new method, he or she should try to locate all the instances of the code pattern and change them to invocations of the new method.

It is usually easy for a developer to locate one or a few places for change, but it is challenging for he or she to locate all the similar places that require the same change. Since the code base of a modern software application is large and usually many developers may have been coding for the application, it may be infeasible for the developer to remember all the similar places without missing any one. Thus, a tool that helps developers locate all the similar places requiring the same change may save much development time.

One possible way to achieve the preceding purpose is to use techniques for code-clone detection. Techniques for code-clone detection are able to detect all the code segments that are similar to the known places that need change. However, techniques for code-clone detection rely on a pre-defined uniform similarity metric to measure the similarity between two code segments. But the similarity in the preceding problem is actually some common characteristics that similar code elements should have, and a different change may imply different common characteristics. For instance, as shown in the first example in Section 2, the two code elements are "similar" for only the change at hand, but should not be similar in any technique for code-clone detection.

Another possible way to achieve the preceding purpose is to use text search. In a typical text-search tool, a developer can represent the common characteristics as a regular expression and search in the code for matched code elements. However, text-search tools cannot always represent common semantic characteristics as regular expressions. For instance, as shown in the second example in Section 2, the common semantic characteristics of the need-to-change code elements are based on control and data dependence relationships, and cannot be represented as a regular expression.

In this paper, we propose a novel approach to locating semantically similar code elements in source code. The basic idea of our approach is as follows. When a developer wants to search for places requiring the same change in the source code, our approach allows him or her to summarize common semantic characteristics of the known places requiring the change and write a query using a language named the Dependence Query Language (DQL) (presented in Section 4.1) to describe the common semantic characteristics. Then, our approach transforms the query into a series of graph patterns and matches the graph patterns in the System Dependence Graph (SDG) of the source code using a fast algorithm for graph-pattern matching [3]. We empirically evaluated our approach using four searching tasks in two real-world projects. Our empirical results demonstrate that our approach is more effective than both code-clone detection and text search for these searching tasks.

This paper makes the following main contributions:

- A demonstration of challenges in locating semantically similar code elements in source code.
- A query language (i.e., DQL) that allows developers to de-

ASE '10, September 20-24 2010, Antwerp, Belgium

scribe common semantic characteristics involving multiple program points and dependence relationships between the program points.

- A search technique that allows developers to locate code elements sharing the same semantic characteristics via graph matching.
- An empirical evaluation demonstrating the effectiveness and the cost of our approach.

We organize the remaining of this paper as follows. In Section 2, we present examples to motivate our approach. In Section 3, we present some background knowledge used in our approach. In Section 4, we present technical details of our approach. In Section 5, we present further optimization of our approach. In Section 6, we report an empirical evaluation of our approach. We discuss related work in Section 7. We conclude this paper with some pointers to future work in Section 8.

2. EXAMPLES

In this section, we present two examples to demonstrate the challenges in locating semantically similar code elements. The first example comes from code changes made to the expat[1] project (cvs version 2002-05-17). In the code changes, the developers want to fix some memory leaks that may happen when a call of the `malloc` function fails. One of the changes made to the code is as below.

Original Code:

```
tag = MALLOC(sizeof(TAG));
if (!tag)
    return XML_ERROR_NO_MEMORY;
tag->buf = MALLOC(INIT_TAG_BUF_SIZE);
if (!tag->buf)
    return XML_ERROR_NO_MEMORY;
```

Cahnged Code:

```
tag = MALLOC(sizeof(TAG));
if (!tag)
    return XML_ERROR_NO_MEMORY;
tag->buf = MALLOC(INIT_TAG_BUF_SIZE);
if (!tag->buf){
    Free(tag);
    return XML_ERROR_NO_MEMORY;
}
```

From the preceding two code portions, we can see that the developers want to find cases where a field (i.e., `tag->buf` in the code) of a struct variable (i.e., `tag` in the code) is initialized after the struct variable is initialized. The aim is to add a `free` function when the initialization of the field is failed so that memory leak can be avoided. The key characteristic is two calls of the `malloc` function: One call is to acquire memory for the entire struct variable and the other call is to acquire memory for a filed of the struct variable. As the name of the struct variable and the name of the field may vary in other places, it is difficult to find these places with code-clone detection. Furthermore, as it is impossible to express the constraint between the struct variable and the field in text search, we cannot use this constraint to confine the search. Actually, another instance of this kind of memory leak is as below.

```
newE = (ELEMENT_TYPE *)lookup(&(newDtd->elementTypes),
   name, sizeof(ELEMENT_TYPE));
if (!newE)
   return 0;
if (oldE->nDefaultAtts) {
   newE->defaultAtts = (DEFAULT_ATTRIBUTE *)
     MALLOC(oldE->nDefaultAtts * sizeof(DEFAULT_ATTRIBUTE));
   if (!newE->defaultAtts)
     return 0;
}
```

[1]The expat project is a popular XML document handling library written in C.

In the preceding code portion, the name of the struct variable is `newE` and the name of the filed is `newE->defaultAtts`. Furthermore, this code portion only depicts one call of the `malloc` function, and the other call of the `malloc` function is inside the implementation of the `lookup` function. To make the situation even worse, there is an extra if-structure (i.e., `if(oldE->nDefaultAtts)`) in this code portion. This example demonstrates that neither code-clone detection nor text search allows developers to express some important constraints when locating semantically similar code elements.

The second example is from code changes in the gpsbabel project (cvs version 2004-10-27). In these changes, the developers try to find copying operations between two arrays of the `UC` type in the form of a loop. The aim is to replace each such copying operation with a call of a specially defined function (i.e., `arraycopy`). The following two code portions depict the code before and after such a change. The code in the italicized parts highlight the change.

Original code:

```
UC* p, str;
...
for(i = 0; i < 10; i++){
    str[i] = *p++;
}
```

Changed code:

```
UC* p, str;
...
arraycopy(str,p,10);
```

The key characteristic is a series of assignments between elements in two arrays and each element is of type `UC`. In this example, the main challenge is to express both the following constraints: 1) the constraint between the loop structure and the content inside the loop structure, and 2) the constraint on the type of the elements in the two arrays. Neither code-clone detection nor text search allows developers to express both constraints in one search.

From the preceding two examples, we have the following observations. First, when locating semantically similar code elements, developers may need to express searching conditions as combinations of constraints of various kinds. Code-clone detection may be inappropriate, since code elements satisfying such a searching condition may not look similar at all. Text search may also be inappropriate, since such a search condition typically contains semantic properties besides textual properties. Second, semantic properties described in such a search condition typically include both control dependencies and data dependencies.

Based on the preceding observations, we propose a new approach to locating semantically similar code elements in source code. Our approach allows developers to expression combinations of textual conditions, data dependence conditions, and control dependence conditions in the individual queries. To deal with the dependence conditions, our approach searches matched instances in the System Dependence Graph (SDG) of the source code.

3. BACKGROUND

3.1 System Dependence Graph

The system dependence graph (SDG) is a graph that describes the dependence relationships between program points in the source code. In an SDG generated by CodeSurfer[2], each node corresponds to a program point in the code. Each program point represents a piece of code. There are 33 types of program points in total in

[2]CodeSurfer is static program analyzer and it can extract a system dependence graph from source code. CodeSurfer can be obtained from http://www.grammatech.com/products/codesurfer.

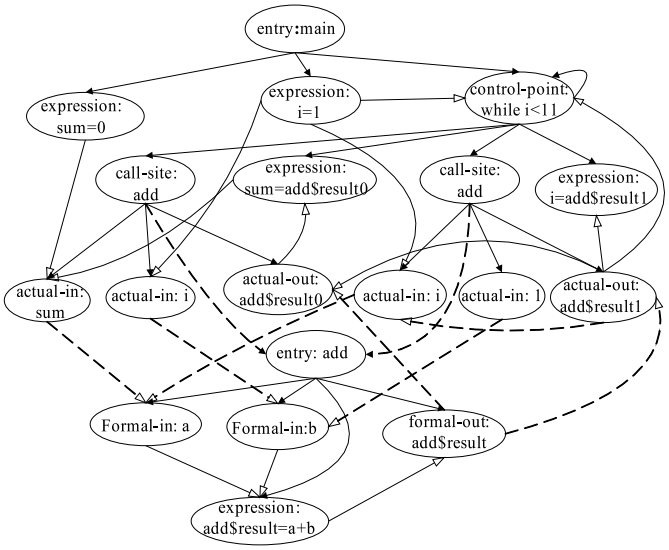

Figure 1: An example system dependence graph

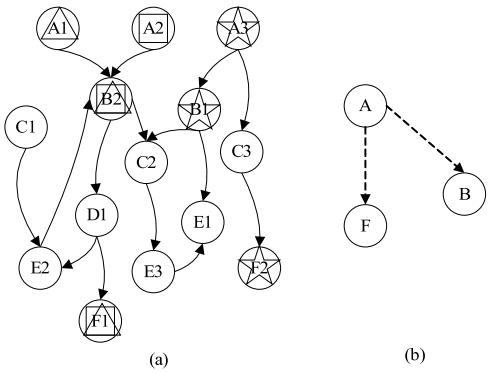

Figure 2: Example Graph and Query

an SDG generated by CodeSurfer. The edges in an SDG correspond to the dependence relationships between code pieces. There are two kinds of dependence relationships: data dependence relationships and control dependence relationships. Table 1 depicts the most commonly used types of program points in SDGs. In our current approach, we only consider searching for the structures consisting of program points of these most commonly used types. However, it is possible to extend our approach to search for other program points by extending our dependence query language (See Section 4.1) and the query transformation rules (See Section 4.3). Note that program points of other types may be intermediate nodes transiting dependence in the graph between program points of the most commonly used types.

Given a C program, CodeSurfer can extract an SDG automatically along with the compilation of the program. For example, for the following program, CodeSurfer can provide an SDG in Figure 1. In the figure, each node is labeled with its program-point type, and we also provide the textual representation of each node. Hollow arrows refer to data dependence and solid arrows refer to control dependence. Solid lines refer to intra-procedure dependence and dotted lines refer to inter-procedure dependence.

```
static int add(int a, int b){
    return a + b;
}
/* Sum 0 through 10. */
void main() {
    int sum, i;
    sum = 0;
    i = 1;
    while ( i<11 ) {
        sum = add(sum, i);
        i = add(i, 1);
    }
}
```

Table 1: Commonly used types of program points

Program Point Type	Description
expression	value assignment or value return
control point	any kind of branch condition (e.g., if, for, while, ...)
call site	function invocation
actual in	actual argument in a function invocation
actual out	returned value of a function invocation
declaration	declaration of a variable
entry	entry of a function

3.2 Graph Reachablity Indexing and Querying

Graph reachability indexing and querying is a research topic in the area of graph databases [3]. A graph reachablity indexing algorithm takes a labeled directed graph as its input. In a labeled graph, each node has one and only one label from a known label set. The algorithm indexes all the reachability relationships between node labels. With the indices of a graph, one can query and obtain all the instances of a reachability pattern (represented as a query graph). For example, for the graph in Figure 2(a) with a label set {A, B, C, D, E, F}, a query in Figure 2(b) can be made. To distinguish nodes labeled with the same label, in Figure 2(a), we add numbers to the end of the label. For example, A1, A2, and A3 are three nodes labeled with A. The query in Figure 2(b) denotes sub-graphs having the following conditions: First, in such a sub-graph, there is at least one node labeled with A, one node labeled with B, and one node labeled with F. Second, in such a sub-graph, it is reachable from the node labeled with A to either the node labeled with B or the node labeled with F. Note that edges between nodes in the query graph represent reachability relations instead of immediate connections. The three matched instances (A1, B2, F1; A2, B2, F1; A3, B1, F2) are marked with triangles, squares, and stars, respectively.

4. APPROACH

To help developers locate semantically similar code elements, we design a language, named the dependence query language (DQL), for developers to describe the common characteristics of target code elements. Section 4.1 presents the details of DQL. With a query written in DQL, our approach searches for code elements that satisfy the common characteristics described in the query using the following four steps. First, we extract the System Dependence Graph (SDG) from the source code using CodeSurfer. Second, we transform the dependence relationships in the query to a series of graph patterns. Third, we search in the SDG to locate sub-graphs matching the graph patterns. Finally, we use the textual constraints in the query to filter the located sub-graphs and trace back to code elements in the source code. Sections 4.2 to 4.5 presents the details of the preceding four steps, respectively.

4.1 Dependence Query Language

A query in our Dependence Query Language has four parts, formally presented as below.

```
Query → QueryDeclaration ; NodeDescriptions ;
        RelationDescriptions ; Wanted
```

The **QueryDeclaration** part is formally defined below.

```
QueryDeclaration → QueryDeclaration , NodeDeclaration
                 | NodeDeclaration
NodeDeclaration → Types Identifier
```

```
Identifier → [A-Z]
Types → Types / Type | Type
Type → function | variable | assignment
     | declaration | control-point
```

This part allows a developer to declare a list of program points involved in the search. Besides an identifier, the developer must give each program point one or more types. If one program point has more than one type, the meaning is to declare a program point whose actual type is one of the types. DQL supports five types of program points: *function*, *variable*, *assignment*, *declaration*, and *control-point*. These five types are abstraction of the types of program points in SDG. We describe the mapping between types in DQL and types in SDG in Section 4.3. The *function* type refers to any invocations of functions. The *variable* type refers to single variables. The *assignment* type refers to assignments or returns of invocations passing the return values[3]. The *declaration* type refers to declarations of variables. As a typical system program dependence graph (e.g., the SDG produced by CodeSurfer) does not represent function declarations as program points, our DQL also does not consider program points representing function declarations. The *control-point* type refers to all branch conditions.

The **NodeDescriptions** part is formally defined as below.

```
NodeDescriptions → NodeDescriptions NodeDescription
                 | NodeDescription
NodeDescription → Identifier Conditions
Conditions → Conditions or Condition | Condition
Condition → not UnitCondition | UnitCondition
UnitCondition → contains String
UnitCondition → declareType String
UnitCondition → declareType Native
UnitCondition → controlType CTYPE
CTYPE → for | while | switch | if
```

In the preceding definition, "String" refers to a character sequence inside quotation marks, "Native" refers to the set of built-in types of the C language, including "char", "float", "int", and "double" etc.

This part allows a developer to describe the properties related to a single program point. For each program point, a developer can add two kinds of properties as conditions. First, a developer can use *contains* to demand the textual representation of a program point to have a particular substring. Second, for certain types of program points, a developer can also describe the declared type or the control type of a given program point. A developer can also add *not* or *or* to the conditions for a program point. Note that our DQL does not support the disjunction of conditions of different program points. In such a case, a developer needs to generate more than one queries.

The **RelationDescriptions** part is formally defined as below.

```
RelationDescriptions → RelationDescriptions ,
                       RelationDescription | RelationDescription
RelationDescription → Identifier op Identifier
op → dependOp | textualOp | structuralOp
dependOp → oneStep subDependOp
subDependOp → dataDepends | controls | calls
textualOp → textual contains
structuralOp → isFieldOf
structuralOp → isElementOf
```

This part allows a developer to describe three kinds of dependence relationships between program points: *dataDepends, controls* and *calls*. Specifically, "A *dataDepends* B" refers to that A is data dependent on B; "A *controls* B" refers to that B is control dependent on A; and "A *calls* B" refers to that there is a chain of function invocations from A to B. A developer can also use modifier *oneStep* before a relationship to demand that the dependence must happen in one step in the SDG.

Our DQL also allow a developer to add textual conditions between the textual representations of two program points. Specifically, "A *textual contains* B" refers to that the textual representation of B is a substring of the textual representation of A.

Sometimes program points may have structural relationships, such as the relationship between a struct variable and its fields or the relationship between an array variable and tis elements. So our DQL also allow a developer add structural conditions. In particular, "A *isFieldOf* B" refers to that A is a field of struct variable B, and "A *isElementOf* B" refers to that A is an element of array variable A. For example, "a.b *isFieldOf* a" holds and "a[b] *isElementOf* a" holds.

The **Wanted** part is formally defined as below.

```
Wanted → want Identifiers
Identifiers → Identifiers , Identifier | Identifier
```

This part allows a developer to indicate which program points in the query are the actual target. For example, in the second example in Section 2, since the target of the developer is only the for/while loops that iteratively assign values to an array of type `UC`, the developer can set only the program points corresponding to the for/while control-point as the wanted program point. In particular, a query for this example is as the following query:

declaration A, control-point B, assignment C; A declareType UC, B controlType for or controlType while; B controls C, C oneStep dataDepends A; want B*

In the preceding query, A, B and C correspond to three program points in the example. A should be a variable declaration and the type of the variable is "UC*", and the node B should be a loop and its control type is "for" or "while". The dependence between C and A is one step because A should directly declares C. B is wanted because the developer wants to locate only the for/while loops.

4.2 SDG Extraction

In our approach, we obtain the system dependence graph (SDG) as follows. First, we use CodeSurfer to generate an initial SDG from the code. Second, we check each node to see whether the node is a program point of one of the types listed in Table 1. For each such node, we extract its type (e.g., *actual-in*, *call-site*, etc.), textual presentation, and location in the code, and label the node with its type. For other nodes, we also label each node with its type but do not extract any information. We keep these nodes in the SDG to transit dependence relationships between nodes of the most commonly used types. Third, we extract all the edges between nodes and label them as control dependence or data dependence. Here we do not differentiate inter-procedure dependence and intra-procedure dependence, since the developer usually does not know whether the target he/she is searching for is in a function or scattered in different functions. Fourth, we give each node in the SDG a unique number as its identification.

4.3 Query Transformation

To use graph reachability querying to search in the SDG, we need to transform a query described in DQL into one or more queries for graph reachability querying. First, as graph reachability querying accept only query graphs, we need to transform queries in DQL into query graphs. Second, as a query graph in graph querying allows only the conjunction of label conditions and reachability conditions, we need to split a query with disjunctions of conditions in DQL into several query graphs. Third, for conditions (such as textual conditions, program-point properties described in the **NodeDescriptions** part, and the *oneStep* conditions) supported by DQL

[3]We merge these two kinds of elements for simplicity, the developer can easily demanding an *assignment* to be or not be a return statement using a textual condition.

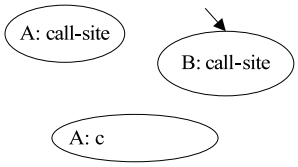

Figure 3: Rules for dependence relation descriptions

by not supported by graph reachability querying, we do not transform them into query graphs but use them to filter the results of graph reachability querying. We present the details of filtering in Section 4.5.

4.3.1 Splitting Queries

In a query written in DQL, there are two places that may contain disjunctions of conditions, and the conditions in both places are conditions for single program points. The first place is the **QueryDeclaration** part, where a program point may have multiple possible types. The second place is the **NodeDescriptions** part, where the properties of a program point may be the disjunction of several conditions. To deal with the disjunctions of conditions, we split the query into the disjunction of a series of sub-queries, each of which contains only conjunctions of conditions. This process is similar to the process of normalizing a logic expression into the disjunctive normal form. When querying the SDL, we use each conjunctive sub-query to obtain a set of results and use the union of all the result sets as the results of the original query.

For example, for the query described as "*function / control-point A, variable B; A contains "abc" or contains "de"; A dataDepends B; want A*", we can split the query into the following four sub-queries: "*function A, variable B; A contains "abc"; A dataDepends B; want A*", "*control-point A, variable B; A contains "abc"; A dataDepends B; want A*", "*function A, variable B; A contains "de"; A dataDepends B; want A*", and "*control-point A, variable B; A contains "de"; A dataDepends B; want A*". None of the sub-queries includes any disjunctions of conditions.

4.3.2 Transforming Conjunctive Queries

For a query containing only conjunctions of conditions, we transform the query into a query graph in the following way. First, we transform each program point in the **QueryDeclaration** part into a node in the query graph. Second, we do not consider the conditions for filtering (e.g., properties described in the **NodeDescriptions** part) in query transformation. Third, we transform the type of each program point into the label of the corresponding node, and the relationships between program points into edges between nodes. Note that there is no straightforward one-to-one mapping between the types of program points in DQL and the types in the SDG and there is no straightforward one-to-one mapping between the relationships in DQL and the relationships in the SDG. In the following, we present the details of transforming program-point types and relationships between program points.

Figure 3 depicts the rules for the preceding transformation. Each rule refers to the transformation of two program points (including their types) and their relationship described in DQL into the corresponding sub-graph (including node labels) in the query graph. In the figure, hollow arrows denote data dependence and solid arrows denote control dependence. Note that some rules result in adding extra nodes into the query graph.

For brevity, in Figure 3, we merge rules when the relationship in DQL is the same and the structure of the transformed sub-graph is similar. As a result, we have four generic rules in Figure 3, and in each generic rule, a program point in DQL may have more than one type and a node in the transformed sub-graph may have more than one label. In such a case, we use a "|" to separate the program-point types and the node labels. The details of the four generic rules are as below.

- ***A* calls *B***. According to the SDG generated by CodeSurfer, such a situation requires that *B* should be control dependent on the entry of *A* and the entry of *A* should be control dependent on the call-site of *A*. So we add two control dependence edges and an intermediate entry node from *A* to *B* (3(a)). Both *A* and *B* should be functions. The newly added node should have an identifier not used by other nodes, and we use *Ai* to indicate that it is the i^{th} added node.

- ***A* controls *B***. We transform this relationship to a control dependence edge from *A* to *B* (Figure 3(b)). *A* should be a control-point, and *B* can be an assignment, a function call or another control-point. Furthermore, if *B* is a function call, we need to transform the *function* type into the *call-site* label. As neither declarations nor references of variables are executable, we do not allow *B* to be a declaration or a variable. We transform the *assignment* type into the *expression* label because the SDG generated by CodeSurfer uses an *expression* to represent an *assignment*.

- ***A* dataDepends *B* and *A* is not a *function***. We transform this relationship to a data dependence edge from *B* to *A* (Figure 3(c)). *A* can be a control-point, an assignment or a variable. As a declaration cannot be data dependent on anything, we do not allow the type of *A* to be *declaration*. *B* can be of any types except for *control-point* (CodeSurfer treats an assignment like "x = a>b" as an expression rather than a control point). Two things need further explanation in Figure 3(c). First, when one program point is data dependent on another program point of the *function* type, the label for the node corresponding to the program point of the *function* type should be *actual-out*. This is because in the SDG generated by CodeSurfer, data dependence to a function is denoted as data dependence to the *actual-out* corresponding to the function call. Note that the dependence between an *call-site* program point and its corresponding *actual-out* is a control

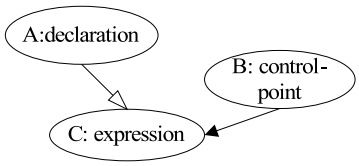

Figure 4: An example query graph

dependence. Second, we transform *variable* into *expression* because the SDG generated by CodeSurfer uses an *expression* to represent an *variable*.

- ***A* dataDepends *B* and *A* is a *function***. In this case, one of the *A*'s actual argument should be data dependent on another program point. Furthermore, the dependencies between a *call-site* and its corresponding actual arguments are control dependencies. So we have the the rule as in Figure 3(d).

Since program points of the *function* type may be transformed into nodes with different labels in different rules. Graph node may have conflicting labels. The possible conflicting labels are *call-site* and *actual-out*. In such a case, we split the node to two, and add a control dependence edge from the *call-site* node to the *actual-out* node.

If no rule in Figure 3 is able to transform a dependence relationship described in a query, we deem that there is an error in the query. For example, if *A* is declared as a *function* and B is declared as a *control-point*, the dependence relationship described as "*A* calls *B*" will result in an error.

As an example, we show the query graph of the query example at the end of Section 4.1 after transformation in Figure 4.

4.4 Graph Querying

As graph reachability indexing and querying requires only one type of edges in both the query graph and the graph for indexing, we cannot use the query graphs to query the SDG. To enable the query, we divide either the query graph or the SDG to two partial graphs. In each partial graph, the nodes are the same with the original graph but we keep only one kind of edges. Thus, after the two partial SDG graphs are indexed, we perform two queries using two partial query graphs on their corresponding partial SDGs.

After we obtain the two lists of results, we need to concatenate the results. We mark a list of split nodes in the query graph when doing the graph division. A split node is a node that has related edges in both partial query graphs. For each result *R1* in the list of results for the control dependence query and each result *R2* in the list of results for the data dependence query, we check whether we can merge all the split nodes[4]. If all the split nodes can be merged, we merge the two results and put the merged result into the list of final results.

4.5 Result Filtering and Generation

After we obtain the results of graph querying, we further use the conditions that are in the original conjunctive query but not transformed into the query graph to filter the results. For each result, we check three categories of conditions. First, we check each node with its corresponding program point in the query. If any property of the program point is not satisfied, we remove the result from the result set. Note that when checking textual presentation of nodes which are labeled as *expression* from a *variable* element. We only use the variable part (e.g., we use only "a" of "a = 0;"). Second, we verify the textual and structural conditions between program points. That is to say, if any two program points do not satisfy any conditions described for them, we remove the result from the result set. Third, we verify the *oneStep* conditions between program points. We examine whether the dependence relationship between two program points is one step by verifying whether there is an corresponding (control or data) edge between the two corresponding nodes in the SDG. For the *calls* relationships, as such a relationship is transformed to multiple edges in the query graph, we examine whether all these edges are one step.

Finally, as a result after filtering may contain several program points, we recommend the developer with only the program points corresponding to the *wanted* part. Furthermore, if the developer wants to examine the reason of our recommendation, we allows the developer to further expand the result and see all the involved program points.

[4]Two nodes in two results can be merged if the two nodes correspond to a same node in the original SDG graph.

5. OPTIMIZATION

In our approach described in Section 4, we use only the types of program points as labels of nodes in the SDG and the query graph. Therefore, we may have a large number of results when querying the SDG, and we need to filter the results using the other conditions described in the query. In some cases, the filtering may become unreasonably time consuming (e.g., more than 24 hours) due to the large number of results in SDG querying.

To solve the problem, we use a pre-filtering technique. The basic idea of pre-filtering is to move the filtering of program-point properties (described in the **NodeDescriptions** part of a query) before the graph querying step. Given an SDG (denoted as G) and a conjunctive query (denoted as Q), we transform Q to a query graph (denoted as QG). Based on G, Q, and QG, our pre-filtering technique works as follows.

For each node (denoted as N) in QG, we assign the node a unique label (denoted as L) and we further find all the nodes in G that match both the type and the properties of N. Note that an extra node added into QG during query transformation has only a type but no other properties. Then, we label all these nodes in G as L. The aim of this change of labels is to constrain the matched nodes of N in G during graph querying. In some cases, one node (denoted as O) in G may match several different nodes in QG. Thus, there is a conflict of label change, since we cannot assign a node in G more than one label. To solve the conflict, we add one or more parallel nodes into G.

For the ease of presentation, let us assume that a node (denoted as O) in G matched two different nodes (denoted as $N1$ and $N2$) in QG and the labels of $N1$ and $N2$ are $L1$ and $L2$, respectively. Thus, we add a new node (denoted as P) into G, such that P has the same neighboring structure as O. Specifically, for each node from which there is an edge to O, there is also an edge to P of the same kind (control or data dependence), and for each node to which O has an edge, P also has an edge to it of the same kind. Therefore, the new node P has the same reachability property with O and adding the parallel node P to G would not affect reachability relationships between any other nodes in G. We assign the label of O as $L1$ and the label of P as $L2$. Thus, during graph querying, $N1$ can match O but not P in G, and $N2$ can match P but not O in G.

Due to added parallel nodes, it is possible that our graph querying algorithm returns a result with two or more nodes corresponding to one original node. In such a result, one node in G is actually matched to two or more nodes in the query graph. As we do not allow this kind of matches, we filter out such matches by checking whether there are two or more nodes parallel with each other in the matched sub-graph.

Table 2: Subjects used in our evaluation

Project Name	Description	Version	Size (KLOC)
expat	xml handle library	2002-05-17	13
		2002-05-22	13
gpsbabel	GPS toolkit	2004-10-27	50
		2005-03-21	54

After using the preceding pre-filtering technique, we need to filter the results of graph querying with only conditions of relationships between program points and the *oneStep* conditions. Our pre-filtering technique can save much time of filtering for our approach, but we need to re-index the SDG when a new query comes. However, according to our experience, the time for indexing is much shorter than that for result filtering without pre-filtering.

6. EMPIRICAL EVALUATION

To evaluate our approach, we applied our approach on four searching tasks in four versions of two different real-world open source C projects: expat and gpsbabel. Table 2 depicts the detailed information about the four versions of the two projects used in our evaluation. To make it easier to generalize the findings of our evaluation, we chose two of the most downloaded C projects in Souceforge[5], and the two projects are of different sizes and from different domains.

To make the four searching tasks close to the real scenario that developers may face, we extracted the four searching tasks from the historical versions of the two projects. In particular, we browsed the historical versions of the two projects, and when we found multiple modifications in one version over its previous version, we checked the comments submitted with the modifications. If the comments imply the application of one change to a number of places, we deemed that the developers need to perform a searching task to apply one change to multiple places.

For each of the four tasks, we built a DQL query based on one changed place that has comments implying the change in multiple places. Section 6.1 presents the results of applying our approach on the four tasks. To help readers understand our results, we also applied an approach based on code-clone detection and an approach based on text search to the same tasks. For the approach based on code-clone detection, we applied a code-clone detector named DECKARD [10] to each of the four versions. We then checked each clone group returned by DECKARD, and we deemed the clone group containing the largest number of places required by the searching task as the results of the approach based on code-clone detection. For the approach based on text search, we built several possible text queries and recorded the results using each text query.

6.1 Results for the Searching Tasks

6.1.1 Task One

The first searching task is actually the first example in Section 2. It is from the expat project version 2002-05-17. The comment of the code change is "*Be more careful about failed MALLOC() and REALLOC() calls. This avoids a number of potential memory leaks*". Example code of the change is shown in Section 2. As we have discussed, the developers want to find places where a field of a struct variable is initialized after the initialization of the struct variable. According to this dependence property, we have the query as below.

function A, function B, variable C, variable D; A contains "malloc" or contains "realloc", B contains "malloc" or contains "realloc"; C dataDepends A, D dataDepends B, C dataDepends D, C isFieldOf D; want C

In the query, *A* corresponds to the initialization of the struct variable, *B* corresponds to the initialization of the field, *C* corresponds to the field, and *D* corresponds to the struct variable. We use *dataDepends* to describe the relationship from the initialization of the variable to the variable itself. We further use *isFieldOf* to describe the relationship between a struct variable and its fields. We put *C* at the wanted list, because *C* is where the change should be made.

In this task, the developers actually changed 2 places (i.e., targets) in the code. Our approach finds both of the 2 places, but our approach also finds 38 other places (i.e., false positives). By contrast, it is not surprising that the approach based on code-clone detection does not find a clone group containing the 2 targets. As shown in Section 2, the two places do not look similar in the sense of code cloning. For text search, the only reasonable way is to search for "malloc" and "realloc" in the code. Actually, there are 33 places in the code containing "malloc" or "realloc". As this task requires two related "malloc" or "realloc", a developer may need to check each pair of the 33 places to know whether the pair forms a target required by the task. Note that one of the 2 targets actually involves two invocations of "malloc()" in two different functions. Not checking this pair of places with "malloc" may result in missing one target.

6.1.2 Task Two

The second searching task is from the expat project version 2002-05-22. The comment of the code change is "*Use "NULL" instead of "0" for NULL pointers. Compare pointers == NULL or != NULL instead of using the implicit point-to-int conversion*". This comment clearly reflects the purpose of the developers (i.e., changing comparisons of pointers with 0 to comparisons of pointers with NULL). Note that this case cannot be caught by the compiler as warnings.

As there are a large number of control points in the program, the essence of this searching task is actually to find all the comparisons of pointers. Therefore, we have the query below.

declaration A, control-point B; A not declareType Native; B oneStep dataDepends A; want B

This query actually finds all the comparisons of variables of non-native types. In the query, *A* corresponds to a declaration of a pointer variable, while *B* corresponds to the control point that involves *A*. The dependence from *B* to *A* is one step because *A* should directly declares the variable involved in *C*. Note that the query is not very specific to the task and may be refined to get better results.

In this task, the developers actually changed 8 places (i.e., targets) in the code. Our approach finds all the 8 targets together with 126 false positives. By contrast, the approach based on code-clone detection still does not find any of the 8 targets. This is also not surprising because the code-clone detector would not deem comparisons of different pointers as cloning. For text search, one possible way is to search for the string "==0" and "!=0" in the code. This search finds only 5 of the 8 targets together with 224 false positives. This search misses 3 targets due to implicit comparisons with "0" when a pointer variable is used as a condition variable (e.g., `p` in `if(p)`). The large number of false positives is due to comparisons between integer variables and "0". Another possible way of using text search is to search for strings like "for", "while", "switch" and "if" in the code, since all the comparisons should appear in these control structures. The second text search finds all the 8 targets, but the number of false positives becomes 837.

[5]http://sourceforge.net/, accessed on 2009.08.31.

6.1.3 Task Three

The third searching task is actually the second example in Section 2 and is from the gpsbabel project version 2004-10-27. The comment of the code change is "*Aggressively replace open-coded strncpy for space padded strings in various waypoint send functions*". The submitted code changes demonstrate that the developers try to find all code structures like the italicized part in the following cod portion, in which a loop is used to copy strings defined by `UC*`, and replace them with a special array copy function.

```
UC* p, str; ...

for(i = 0; i < 10; i++){
    str[i] = *p++;
}
```

The preceding code structure has two dependence relationships. First, there is a loop (either a for loop or a $while$ loop). Second, inside the loop, there is an assignment and the assignment data depends on a declaration that declares a variable with type `UC*` used in the assignment. As we have introduced in Section 4.1, we have the query as below.

declaration A, control-point B, assignment C; A declareType UC, B controlType for or controlType while; B controls C, C oneStep dataDepends A; want B*

In this task, there are actually 39 targets in the code. Our approach finds all the 39 targets together with 110 false positives. By contrast, the approach based on code-clone detection finds 38 targets with 255 false positives. Unlike the previous two tasks, the targets in this task share a common loop structure that can be caught by the code-clone detector. However, the code-clone detector also catches some other loop structures as similar and brings in more false positives. The one target that code-clone detection fails to find is also a for loop. But there are several other statements in the for loop, so that the similarity between the lost target and the other targets drops below the threshold. For text search, one possible way is to search for "for" and "while" in the code. This search finds all the 39 targets, because all the targets involve a for loop or a $while$ loop. But this search also finds 297 false positives, because there are many for loops or $while$ loops used for other purposes. Another possible way is to search for `UC*` in the code and find all the variables of type `UC*`. After that a developer needs to further locate all the references to the these variables to check the assignment to these variables in for or $while$ loops. In total, the developer needs to check more than 2000 references to the variables with type `UC*` in the code.

6.1.4 Task Four

The fourth searching task is from the gpsbabel project version 2005-03-21. The comment of the code change is "*Call waypt_new instead of explicit calloc to prepare for external alt invalid indicator*". In the submitted changes, the aim is to change all `xcalloc()` function calls that initialize variables of type `waypoint*` to calls of a new function (i.e., `waypt_new()`) that specially initializes `waypoint*` variables. An example of such a code element is as below.

```
waypoint *wpt_tmp;
...
wpt_tmp = xcalloc(sizeof(*wpt_tmp),1);
```

In the preceding code element, there are two dependence relationships. The first is data dependence between the `waypoint *wpt_tmp` declaration and the `wpt_tmp` variable. The second is data dependence between `xcalloc()` and `wpt_tmp`. Thus we have the query as below.

Table 3: Execution time of our approach (in seconds)

Task	Query Build	Pre-filter	Index	Query	Concate-nation	Result Filter	Result Merge
task 1	<0.1	1.7	63.7	1.0	N/A	2.1	<0.1
task 2	<0.1	0.9	29.4	<0.1	N/A	1.1	N/A
task 3	<0.1	3.3	640.0	0.2	4.2	3.6	<0.1
task 4	<0.1	3.3	264.4	1.2	N/A	0.9	N/A

declaration A, variable B, function C; A declareType waypoint, C contains "xcalloc"; B oneStep dataDepends A, B oneStep dataDepends C; want C

In the query, *A* corresponds to the declaration of a variable with the `waypoint*` type. *B* corresponds to the variable with the `waypoint*` type. *C* corresponds to the invocation of `xcalloc`. The dependence between *B* and *C* is of one step because variable *C* should be the variable that first holds the value of `xcalloc()`. The dependence between *B* and *A* is of one step because *A* should directly declares *B*. *C* is wanted because the developer wants to locate only `xcalloc()` calls.

In this task, there are actually 19 targets in the code, and our approach finds all the targets together with 3 false positives. By contrast, the approach based on code-clone detection does not find any of them. The reason is that the code-clone detector would not deem invocations of `xcalloc()` as cloning. In text search, one possible way is to search for "xcalloc" in the code. This search finds all the 19 targets, but concedes 86 false positives. The reason is that there are many other places that invoke `xcalloc()`. Another possible way is to search for "waypoint" in the code, and further check all the references of variables with type `waypoint*`. This search returns 620 references to variables with type `waypoint*`.

6.2 Execution Time

We also recorded the execution time of each step in our approach for performing each of the four tasks. Table 3 depicts the results on execution time. Note that we only present the execution time of steps that are involved in performing a query. We do not present the execution time of the preparation steps including compiling the projects, extracting SDGs, and dividing SDGs to two partial SDGs (each one has only one kind of edge). The execution time of the preparation steps are not very important because once the these steps are performed, the developers can do any number of queries without performing these steps again. Actually, for each searching task, the total execution time of all preparation steps is less than one hour. In Table 3, there are three things to be noted. First, the execution time of "Query Build" (column 2) is the total execution time of splitting a query to conjunctive queries (if query splitting is required), transforming from conjunctive queries to graphs, and dividing graphs to partial graphs that contain only one kind of edges (if graph division is required). Second, the execution time of "Concatenation" (column 5) is the execution time of concatenating the results of two partial queries. Since queries for some tasks only contain one kind of dependence relationships, our approach did not perform graph division and concatenation and thus the corresponding execution time for concatenation is not available. Third, the execution time of "Result Merge" (column 7) is the execution time of merging the results of conjunctive sub-queries. Similarly, if a query does not contain disjunctives of conditions and thus result merging is not required, its execution time of result merging is not available.

From this table, we can see that the execution time for the shortest search is around half a minute, and that for the longest search is around 11 minutes. The execution time may be too long for developers to use our approach in an interactive manner. However,

as our approach does not require user involvement in the searching process, it is possible for developers to use our approach in an off-line way. Therefore, the execution time should not be a big burden for developers.

Furthermore, this table also demonstrates that the most time-consuming step in our approach is graph indexing. Currently, since our approach adds some parallel nodes in the pre-filtering step, our approach needs to re-index the graph when a new query comes. As the number of added nodes is small, it should be possible to develop an incremental indexing algorithm to add more indexing information about the added parallel nodes on the base of the existing indexing information. With such an incremental indexing algorithm, we may further accelerate our approach.

6.3 Threats to Validity

The main threat to internal validity in our evaluation is the possible faults in the implementation of the evaluated approaches. To reduce this threat, we acquired the implementation of DECKARD from one author of the tool and used the text-search facility of a mature text editor. Furthermore, we reviewed all the code of the implementation of our approach before conducting the evaluation.

In our evaluation, we applied our approach with the approach based on code-clone detection and the text-search approach on four searching tasks for two subjects. This factor may be a threat to the external validity, as our empirical results may be specific to the used tasks and subjects and thus not generalizable. To reduce this threat, we used different tasks with subjects from different domains. Further reduction of this threat needs further evaluation with more tasks and more subjects.

As the searching tasks were recovered from the subjects, the scenarios involved in the tasks may not reflect scenarios in real-world development. This factor may be a threat to the construct validity. To reduce this threat, we used the version history of each subject, and carefully examined the change comments and the changed code to figure out the purpose of the developers for each searching task.

6.4 Discussion

According to the results presented in Section 6.1, our approach has a considerable number of false positives in the four searching tasks. These false positives are mainly due to that we used simplified ways to specify search requirements. However, compared with code-clone detection and text search, in which many search requirements cannot be specified, our approach does provide a means to appropriately specify different search requirements. In fact, it is still possible to refine the queries for our approach to further reduce the false positives.

Furthermore, the searching tasks imply that the developers know what the targets should look like. That is to say, the developers can use our approach for real-world development in spite of the false positives, since it is possible for they to check each result returned by our approach to distinguish targets from false positives. Note that our approach is able to find all the targets in all the tasks with much fewer false positives than the other two approaches in our evaluation.

7. RELATED WORK

In essence, our approach searches for code elements satisfying some common semantic constraints described with our DQL. Thus, the research most related to our approach is various generic code-search techniques. Generic code-search techniques are code-search techniques that allow users to write different queries for different searching tasks. The main advantage of generic code-search techniques is that users can easily use them in different searching tasks without changing the underlying searching algorithm.

The most popular category of generic code-search techniques is text search, including plain text search, regular expression search, and natural language based code search [6, 9]. The main difference between these techniques and our technique lies in that, besides textual information, our approach further allows users to describe dependence relationships between program points in the target code elements. Our empirical results demonstrate that descriptions of dependence relationships help improve the accuracy of code search in many searching tasks.

Another category of generic code-search techniques is code-search based on model checking. By using mode checking, such a technique allows users to describe temporal conditions in queries. For example, Li and Zhou used model checkers to locate the outliers of coding rules, such as finding the place where a call to `File.open()` is not followed by a call `File.close()` [12]. Another typical example in this category is a technique recently proposed by Brunel et al. [2]. As a supporting technique for a generic patch-inference tool [1] (which infers and applies common characteristics of patches for Linux drivers), Brunel et al.'s technique can locate code elements satisfying certain control-flow properties using model checking. Unlike Brunel et al.'s technique, our technique uses graph matching to check which code elements satisfy the required semantic characteristics. Furthermore, our approach allows users to describe data dependence relationships as well as control dependence relationships between program points in queries. In some searching tasks discussed in this paper, description of data dependence relationships in queries is essentially helpful for locating semantically similar code elements. Actually, as data dependence relationships in the System Dependence Graph (SDG) may not be described as temporal relationships, model checkers may not be suitable to check these data dependence relationships.

Generic code search can also be based on checking properties acquired at runtime. Martin et al. proposed the Program Query Language (PQL) [14], which allows users to describe a pattern of sequentially executed invocations on objects. For each query, the searching technique based on PQL first statically checks the program to acquire a list of candidate program points that may appear in instances of the pattern. Then the candidate program points are instrumented and the pattern is matched dynamically during the execution of the instrumented program. There are three main differences between the search technique based on PQL and our technique. First, PQL focuses on control flows and does not allow descriptions of data dependence relationships. Second, PQL focuses on only invocations of objects, while our technique can also handle many other program points, such as expressions, control points, and declarations. Third, PQL matches code patterns dynamically, while our technique matches queries statically.

To our knowledge, our technique is the first generic code-search technique that allows users to describe data dependence relationships, control dependence relationships, and textual conditions in one query.

The main disadvantage of generic code-search techniques is that they may not be sufficiently precise on some particular tasks. Therefore, there are also intensive investigations on specialized code-search techniques for some commonly encountered tasks. Specialized code-search techniques are code-search techniques that search for code elements with pre-defined patterns. The searching algorithm in a specialized code-search technique typically focuses on just one pre-defined pattern and thus can be more precise, but no specialized code-search techniques can search for a pattern other than the pre-defined patterns. That is to say, generic code-search techniques and specialized code-search techniques are not replaceable with each other.

Code-clone detection [11, 10, 5] is a category of techniques that focus on finding similar code segments in the code base of a program. Using code-clone detection, developers can search for code portions with certain properties (that some code portions are known to share). Since developers typically have known some code elements when locating all the semantically similar code elements, code-clone detection may be adopted for this purpose. However, our empirical results demonstrate the superiorness of our approach over code-clone detection for this purpose. We suspect the essential reason to be that code-clone detection techniques are actually specialized code-search techniques designed for another purpose.

Slicing is a category of techniques that locate a subset of the program points that affect a certain variable or depend on a certain variable [19]. In this sense, it is also a specialized code search technique. The main difference between our technique and slicing is that slicing must start from a known variable, and the query is pre-defined (all program points affecting or be affected by the variable), while our technique allows more flexible queries defined by the developers.

Our work is also related to techniques for recommendation of code samples [7, 21], which actually search for matched code samples in the source code of a number of programs. Our technique differs from these techniques in the following ways. First, these techniques work on only API-invocation structures, while our work can handle both API invocations and other program points. Second, these techniques use pre-defined rules, and users cannot describe the characteristics of target code samples in queries.

FindBugs[6] [8] and PMD[7] [4] are two popular static checking tools. Both tools use pre-defined bug patterns and can search for instances of these bug patterns in source code. For each pattern, a detector, which can be integrated with the two tools based on some code managing APIs, is developed specially to search for code elements matching the pattern. However, for the scenario of locating semantically similar code elements, the pre-defined patterns may not be sufficient for all cases. It is not feasible for developers to write a new detector for each new searching task.

Additionally, there are a number of efforts on locating code elements in more specialized tasks. Due to space limit, we only introduce some recent efforts as below. Maule et al. proposed a technique to search for code elements that are affected by a given database-schema change [15]. Wang et al. proposed a technique to search for constant string variables that finally goes to the GUI to facilitate software internationalization [18]. These two techniques are essentially specialized code-search techniques, because they are specific to two kinds of code patterns. In a broader sense, feature-location techniques [20] can be viewed as specialized code-search techniques, because such a technique tries to search for code elements that are related to a given feature. Jungloid-search techniques [13, 16, 17] can also be viewed as specialized code-search techniques, because such a technique tries to search for conversion code from a source object type to a target object type. A distinct feature of Jungloid-search techniques is that the search is not confined to the source code of just one program.

8. CONCLUSION AND FUTURE WORK

In this paper, we propose a novel approach that helps developers locate semantically similar code elements. Our approach includes a query language that allows developers to describe dependence relationships, an algorithm to transform queries to graph patterns, a graph-indexing and querying algorithm, a result filter, and an optimizer based on pre-filtering. We evaluate our approach on four searching tasks acquired from four versions of two real-world C projects. The results of our evaluation demonstrate that our approach is able to locate all the targets with reasonable execution time. Furthermore, our approach outperforms code-clone detection and text search by finding more targets and conceding fewer false positives.

There are several ways to improve or extend our approach. First, as mentioned in Section 6.3, there are several threats to the validity of our empirical results. We plan to expand the set of searching tasks by studying more versions of more projects. Furthermore, we also plan to do some user studies in real-world projects to further evaluate our approach.

Second, in this paper, we use graph indexing and querying to search the system dependence graph that describes dependence relationships between program points. As there are also other kinds of graphs (e.g., class diagrams in object-oriented programs) in software development, we plan to extend our approach for these graphs.

Third, our pre-filtering requires re-indexing the system dependence graph when a new query comes and the re-indexing is the most time-consuming part in our approach. As re-indexing in our approach actually faces a small number of added nodes, we plan to investigate incremental re-indexing algorithms that can index only the newly added nodes.

[6]http://findbugs.sourceforge.net/

[7]http://pmd.sourceforge.net/

9. REFERENCES

[1] J. Andersen and J. L. Lawall. Generic patch inference. In *ASE*, pages 337–346, 2008.

[2] J. Brunel, D. Doligez, R. R. Hansen, J. L. Lawall, and G. Muller. A foundation for flow-based program matching: using temporal logic and model checking. In *POPL*, pages 114–126, 2009.

[3] J. Cheng, X. J. Yu, B. Ding, P. Yu, and H. Wang. Fast graph pattern matching. In *ICDE*, pages 913–922, 2008.

[4] T. Copeland. *PMD Applied*. Centennial Books.

[5] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE*, pages 321–330, 2008.

[6] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of NL-queries for software maintenance and reuse. In *ICSE*, pages 232–242, 2009.

[7] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE*, pages 117–125, 2005.

[8] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[9] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *AOSD*, pages 178–187, 2003.

[10] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, 2007.

[11] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *TSE*, 28(7):654–670, 2002.

[12] Z. Li and Y. Zhou. PR-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE*, pages 306–315, 2005.

[13] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Jungloid mining: Helping to navigate the API jungle. In *PLDI*, pages 48–61, 2005.

[14] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: A program query language. In *OOPSLA*, pages 365–383, 2005.

[15] A. Maule, W. Emmerich, and D. S. Rosenblum. Impact analysis of database schema changes. In *ICSE*, pages 451–460, 2008.

[16] N. Tansalarak and K. T. Claypool. XSnippet: Mining for sample code. In *OOPSLA*, pages 413–430, 2006.

[17] S. Thummalapenta and T. Xie. ParseWeb: a programmer assistant for reusing open source code on the web. In *ASE*, pages 204–213, 2007.

[18] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun. Locating need-to-translate constant strings for software internationalization. In *ICSE*, pages 353–363, 2009.

[19] M. Weiser. Program slicing. In *ICSE*, pages 439–449, 1981.

[20] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a static noninteractive approach to feature location. *TOSEM*, 15(2):195–226, 2006.

[21] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *ECOOP*, pages 318–343, 2009.