Graph-based Detection of Library API Imitations

Chengnian Sun*, Siau-Cheng Khoo*, Shao Jie Zhang†
*School of Computing, National University of Singapore
†NUS Graduate School for Integrative Sciences Engineering , National University of Singapore
{suncn, khoosc}@comp.nus.edu.sg, shaojiezhang@nus.edu.sg

***Abstract*—It has been a common practice nowadays to employ third-party libraries in software projects. Software libraries encapsulate a large number of useful, well-tested and robust functions, so that they can help improve programmers' productivity and program quality. To interact with libraries, programmers only need to invoke Application Programming Interfaces (APIs) exported from libraries. However, programmers do not always use libraries as effectively as expected in their application development. One commonly observed phenomenon is that some library behaviors are re-implemented by client code. Such re-implementation, or imitation, is not just a waste of resource and energy, but its failure to abstract away similar code also tends to make software error-prone. In this paper, we propose a novel approach based on *trace subsumption* relation of data dependency graphs to detect imitations of library APIs for achieving better software maintainability. Furthermore, we have implemented a prototype of this approach and applied it to ten large real-world open-source projects. The experiments show 313 imitations of explicitly imported libraries with high precision average of 82%, and 116 imitations of static libraries with precision average of 75%.**

I. Introduction

A software library is a collection of subroutines or classes to facilitate software development. A library contains code and data which provide certain services and can be exported via Application Programming Interfaces (APIs) to independent programs. (We refer to the code that relies on libraries as *client*). Libraries provide a cost effective way to build software systems: they improve the productivity of programmers by providing a variety of desired functionalities; they enhance software quality as libraries are usually well-tested and thus fairly robust thanks to their massive and diverse user base. Nowadays, it has been a common practice to enjoy the benefits of libraries in software projects large and small.

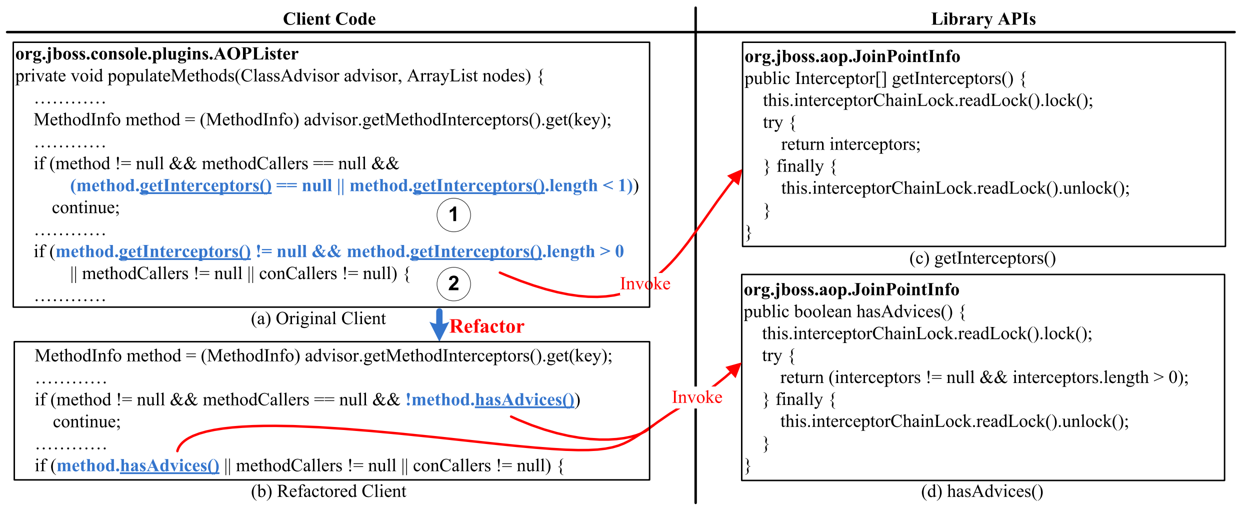
Unfortunately, libraries are not always effectively used by programmers in developing their applications. In particular, we often find client code that *imitates* library API, *i.e.*, client code re-implements the behavior of a library API. There are several reasons for such imitations which are also elaborated in [1], *e.g.*, programmers might not be familiar with the library, not aware of all the functionalities, or lost in a huge collection of APIs. Reinventing the wheel does not only waste unnecessary resources, time and energy, but its failure to abstract away similar code also tends to make software error-prone. This maintenance concern has motivated us to find a tool that can detect imitations of library APIs in client code both efficiently and effectively.

Figure 1 is a simple imitation example we found in J-Boss application serverV5.0.1. Class *MethodInfo* is a subclass of class *JoinPointInfo*. Each link emitting from client code represents the client invoking a library method. As shown in Figure 1(a), the method $populateMethods$ first tests the expression $(method.getInterceptors() == null \;||\; method.getInterceptors().length < 1)$; let's call it e. Then it tests the negation of e in the second if-condition. Details of library method $getInterceptors$ are given in Figure 1(c). It firstly acquires the lock $readlock$, returns its interceptors to the client and lastly releases the lock. It is not difficult to find out that the library method $hasAdvices$ (shown in Figure 1(d)) has the same functionality as testing e. Thus if the method $hasAdvices$ is known beforehand, programmers may replace each appearance of e with $hasAdvices$ as Figure 1(b) shows. As a result, the original code is refactored into a more succinct and efficient fashion, which alleviates two method calls and two pairs of locking operations.

Straightforward though this imitation may seem, it is not easily identifiable by usual text-based similarity detection techniques. The reason is that the lock operations only appear in the library API $hasAdvices$ and in the method $getInterceptors$ called in the client code. The imitation becomes clear only when the implementation of $getInterceptors$ is explicitly inlined into the client code.

In this work we take a semantic stance to investigate the problem of detecting such imitation. Here we focus on the similarity between the *data dependency graphs* (DDGs) of a coding pair consisting of a library API and a client method. Generally speaking, the DDG of a library API (G_{lib}) is said to be imitated by the DDG of a client method (G_{clt}) if and only if all the data flow traces of G_{lib} can be "subsumed" by those of G_{clt}. If G_{clt} imitates G_{lib}, then the client method potentially imitates the library API.

Central to our approach is a novel algorithm for detecting graph imitation based on *trace subsumption* relation. We first present a list of matching rules to identify trace subsumption relation between the data flow traces of both a client method and a library API. Then we propose several effective heuristic techniques to prune method inlinings thus achieving better scalability. Lastly, we implemented a prototype of our approach and applied it to ten open-source Java projects which were also selected by Kawrykow and Robillard [1]. We first experimented on detecting imitation of codes from the set of library APIs which have been explicitly imported in client code. Our method reported 313 true imitations, with



```
org.jboss.console.plugins.AOPLister
private void populateMethods(ClassAdvisor advisor, ArrayList nodes) {
  …………
  MethodInfo method = (MethodInfo) advisor.getMethodInterceptors().get(key);
  …………
  if (method != null && methodCallers == null &&
      (method.getInterceptors() == null || method.getInterceptors().length < 1))
    continue;
  …………
  if (method.getInterceptors() != null && method.getInterceptors().length > 0
      || methodCallers != null || conCallers != null) {
  …………
```

(a) Original Client

```
  MethodInfo method = (MethodInfo) advisor.getMethodInterceptors().get(key);
  …………
  if (method != null && methodCallers == null && !method.hasAdvices())
    continue;
  …………
  if (method.hasAdvices() || methodCallers != null || conCallers != null) {
```

(b) Refactored Client

```
org.jboss.aop.JoinPointInfo
public Interceptor[] getInterceptors() {
  this.interceptorChainLock.readLock().lock();
  try {
    return interceptors;
  } finally {
    this.interceptorChainLock.readLock().unlock();
  }
}
```

(c) getInterceptors()

```
org.jboss.aop.JoinPointInfo
public boolean hasAdvices() {
  this.interceptorChainLock.readLock().lock();
  try {
    return (interceptors != null && interceptors.length > 0);
  } finally {
    this.interceptorChainLock.readLock().unlock();
  }
}
```

(d) hasAdvices()

Fig. 1. A Library API Imitation Example in JBoss Application Server V5.0.1

precision average of 82%. This is an improvement over the work by Kawrykow and Robillard [1]. We next experimented on detecting imitation of utility methods available in static library APIs. Here, our method reported 116 true imitations with precision average of 74%.

We summarize our contributions as follows:

1) We employ data dependency graphs – a semantics-based representation of programs – as the basis for detecting imitations.
2) We propose a novel graph algorithm based on trace subsumption relation to detect imitations of library APIs in client code.
3) We demonstrate the practicality of our method by detecting imitations with high precisions.
4) Our approach is scalable and can complete detection for large software projects within 130 seconds.

The paper is organized as follows. Section II presents a brief introduction to data dependency graphs and method inlining. Section III details our technique including the basic definitions, algorithms and heuristics to detect library imitations. Section IV describes our case studies on sizable open source projects and shows the effectiveness of the proposed approach in terms of performance. Section V discusses some important considerations about our approach. Section VI reviews related work, and finally, Section VII concludes and describes some potential future work.

II. Preliminaries

In this section, we introduce data dependency graphs (DDGs), which capture the data dependencies among program statements. Then we present the concept of method inlining which helps detecting imitations at DDG level.

A. Data Dependency Graph

A *data dependency graph* [2] [3] is a graphical representation of a method which consists of vertices and directed edges. Each vertex represents a basic statement (usually in three-address code form) such as a method call, an arithmetic operation and a field read or write. An edge $v \rightarrow u$ denotes that the vertex u is data-dependent on v. Data dependency is formalized as follows.

*Definition 1 (**Data Dependency**):* A vertex u is *data dependent* on a vertex v if a variable var defined at v is used at u and there is an execution path P from v to u, along which the var is not re-defined.

*Definition 2 (**Data Dependency Graph**):* The *data dependency graph* of a method M is a triple $G = (V, E, L)$, where

- V is a set of vertices, each of which represents a basic statement in M,
- $E \subseteq V \times V$ is a set of data dependency edges between statements,
- $L : V \rightarrow T$ is a function that assigns vertex types to vertices.

Table I lists all vertex types considered in our approach. *E.g.*, each formal parameter of a method is a vertex of type *Formal*. If a DDG is extracted from an instance method, there is a vertex of type *This* representing the current object instance. Further, in order to track how an actual parameter value is used in a method call, we introduce two additional vertices *Actual* and *Callee*. For each call C to a method M, each formal parameter of M has a vertex of type *Actual* representing the corresponding actual parameter preceding the *Call* vertex of C. If M is an instance method, there is also a *Callee* vertex preceding the *Call* vertex of C. Likewise, a *RtnVal* vertex is added to the end of C representing the return value of M if the value is not void.

*Definition 3 (**Value Vertex**):* A vertex is a *value vertex* if and only if its statement defines a value.

Each vertex with an attribute *type* is a value vertex in Table I.

*Definition 4 (**Entry**):* An entry of a DDG is a vertex without sink edges.

TABLE I
VERTEX TYPES AND THEIR ATTRIBUTES IN DDG

Type	Description	Attributes
Actual	An actual parameter of a method call	*index*: parameter index
ArrIdx	A read operation from an element in an array	*type*: type of the element
Binary	A binary arithmetic or logic expression	*type*: type of the expression value; *op*: $+/-/\times/\div/\wedge/\vee/\cdots$
Call	A call to a method M	*cls*: declaring class of M; *sig*: signature, ie., name + parameter type list
Callee	The receiver of an instance method call	
Cond	The conditional in a branching statement	
Const	A primitive, string or null constant	*type*: constant type; *value*: constant value
FGet	A read operation from a field	*type*: field type; *cls*: declaring class ; *name*: field name
FPut	A write operation to a field	*cls*: declaring class ; *name*: field name
Formal	A formal parameter of a method	*type*: parameter type; *index*: parameter index;
Exit	The return statement of a non-void method	
NewArr	An invocation to create an array	*type*: type of the array
NewObj	An invocation to an object construction	*type*: type of the constructed object; *params*: parameter type list
RtnVal	The return value of a method call	*type*: type of the return value
This	The reference to the current object	*type*: type of the current object

An entry can be of type *NewObj* with no parameters, *Call* to a static method with no parameters, *FGet* to a static field, *Const*, *This*, *Formal*, *etc.* There may be multiple entries in one DDG.

Figure 2 shows the DDG of method $hasAdvices$ given in Figure 1(d). Underlined text denotes vertex types. There are three entries $This$, 0 and $null$. The vertex $This$ is the reference to the current object, as the method is an instance method. Succeeding $This$ are four field reads. Preceding the $Exit$ vertex is the AND logical operation.

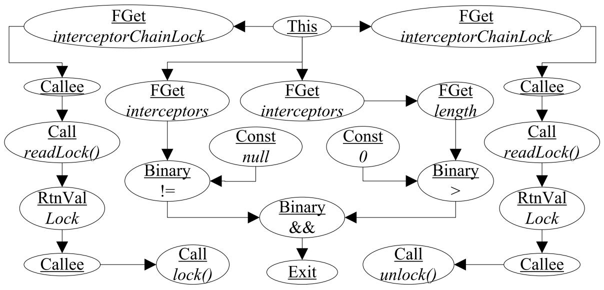


Fig. 2. The Data Dependency Graph of $JoinPointInfo.hasAdvices()$

B. Method Inlining

Method inlining plays a vital role in detecting imitations of library APIs as imitation instances are not often simple copy-and-paste clones as shown in the motivating example. Suppose the DDG of a caller method is G_{caller} and the DDG of a callee method is G_{callee}. V_{call} is the corresponding *Call* vertex. We define the following rules to inline G_{callee} into G_{caller}:

- **Actual Parameters**: We add edges from the predecessor of each i-th actual parameter vertex V_{actual} to each successor of the i-th formal parameter vertex V_{formal} of G_{callee} and remove V_{actual}, V_{formal} and their associated edges.
- **Callee**: We add edges from the predecessor of the $Callee$ vertex V_{callee} in G_{caller} to each successor of the $This$ vertex V_{this} in G_{callee} and remove V_{callee}, V_{this} and their associated edges.
- **RtnVal**: We add edges from each predecessor of the *Exit* vertex V_{exit} in G_{callee} to each successor of the *RtnVal* vertex V_{rtn} in G_{caller} and remove V_{rtn}, V_{exit} and their associated edges.

III. APPROACH

Our approach to detecting client code snippets which imitate behaviors of library APIs is based on data dependency graphs. We propose a *trace subsumption* relation to determine the similarity between data flows in library and client methods. Given a library API lib and a client method clt, if all the data flows in lib can be subsumed by clt, then we assert that clt is potentially imitating the behavior of lib.

In this section we begin by introduing the relevant notions and terminologies. Then we present the workflow of our approach in detail, including pre-processing library APIs, imitation checking algorithms and post validation of imitations.

A. Basic Definitions

*Definition 5 (**Trace**):* A *trace* in a data dependency graph is a sequence of vertices such that from each of its vertices except the last one there is an edge to the next vertex in the sequence, and the first vertex in the sequence is an entry of the graph.

Given a data dependency graph G, let V be its vertex set. We denote a *trace* T in G as $\langle v_1, v_2, ..., v_{end}\rangle$ where each v_i is a vertex from V and v_1 is an entry. We also use the notation $traces(G)$ to represent all the (possibly infinite) traces of G.

A trace captures the data flow from the beginning of a method to some statements within the method. Considering a trace in Figure 2,

$$\langle \textit{This, FGet:interceptorChainLock, Callee, Call:readLock(), RtnVal:Lock, Callee, Call:unlock()} \rangle$$

the current object *This* is first used to retrieve the content of the field $interceptorChainLock$, on which $readLock()$ is then invoked. Finally $unlock()$ is called on the returned $Lock$ object. We believe that in most cases where the data flows in a library API are similar to those in client code, the client code does imitate library APIs.

Detecting similarity between data flows requires us to compare vertices from two DDGs. A set of matching rules is listed in Table II. We define a predicate $match(v, u)$ to test whether the vertex v in a library DDG matches u in a client DDG. $match(v, u)$ returns $true$ if we can find a row such that the first column is the type of v, the second column is the type of u and the evaluation of the formula in the third column yields $true$. Most of the evaluation criteria are based on the equality of types and attributes of two vertices, except for *Formal*, *This*, *Binary*, *Call*, *Const* and *Exit*. The rules for *Formal* and *This* naturally follow the type constraint of method invocations, *i.e.*, a value of type V can be passed to a method call as a parameter or callee of type F if and only if V is F or a subtype of F. As developers can write a program in various ways, We relax the matching criteria for *Binary* and *Const* (*cf.* the first and third rows in Table II). This relaxation helps harvesting more imitations such as the motivating example at the cost of few false positives. In Figure 1(a), although the first highlighted conditional uses operators different from those in $hasAdvices()$, our implementation can still detect it.

TABLE II
MATCHING RULES: $match(v, u)$

v	u	**Condition**
Binary	Binary	$v.type = u.type \wedge v.op \approx u.op$[1]
Call	Call	$(u.cls \leq: v.cls \vee v.cls \leq: u.cls)$[2] $\wedge$ $v.selector = u.selector$
Const	Const	$v.type = u.type \wedge v.value \approx u.value$[3]
Formal	Value[4]	$u.type \leq: v.type$
Exit	$*$[5]	true
This	Value[4]	$u.type \leq: v.type$
Default		u and v are of the same vertex type, and all attributes are the same

[1] $v.op \approx u.op$: $true$ if the operators of v and u are in the same operator category. There are six operator categories $\{+, -\}$, $\{\times, \div\}$, $\{mod\}$, $\{=, \neq\}$, $\{other\ logic\ operators\}$ and $\{others\}$
[2] $a \leq: b$: a is b or a subtype of b;
[3] $v.value \approx u.value$: true if (1) both are boolean, (2) or both are numbers and $|v.value - u.value| \leq 1$, (3) or $v.vlaue = u.value$
[4] *Value* represents a *value* vertex defined in Definition 3.
[5] $*$ denotes any vertex type.

Based on the matching rules, we define a *subtrace* relation of two traces as follows.

*Definition 6 (**Subtrace Relation**):* A trace $t_1 = \langle v_1, v_2, ..., v_m \rangle$ is considered a *subtrace* of another trace $t_2 = \langle u_1, u_2, ..., u_n \rangle$ if there exists an integer i, $0 \leq i \leq n - m$ where $match(v_1, u_{1+i})$, ... and $match(v_m, u_{m+i})$. Notation-wise, we write this relation as $t_1 \leq t_2$.
Intuitively, $subtrace$ is a generalization of $substring$ relation. $Substring$ is based on the equality of characters in strings, whereas $subtrace$ is based on the matching rules of vertices.

We use the *trace subsumption* relation to decide whether the data flows in a DDG are imitated by those in another DDG.

*Definition 7 (**Trace Subsumption Relation**):* Given two data dependency graphs G_{lib} and G_{clt}, G_{clt} *trace subsumes* G_{lib}, if and only if for each trace $t_1 \in traces(G_{lib})$ there exists at least one trace $t_2 \in traces(G_{clt})$ such that t_1 is a subtrace of t_2. Notation-wise, we denote this relation as $G_{lib} \sqsubseteq G_{clt}$. Alternatively, it can be formally defined as:
$G_{lib} \sqsubseteq G_{clt} \equiv$
$$\forall t_1 \in traces(G_{lib}) : \exists t_2 \in traces(G_{clt}) \wedge t_1 \leq t_2$$
Similar to [1], as some of the library imitations are not simple copy-paste code clones, we need to inline some method(s) into the client or library DDG or both if necessary. For practicality, we elect to perform inlining only on the original method code, and not the inlined code. Based on the definition of *trace subsumption* relation, the following definition identifies the principle of our approach.

*Definition 8 (**Potential Imitation**):* A client method clt is said to *potentially imitate* a library method lib, if after inlining some method calls in clt and/or lib, the resultant DDG of clt *trace subsumes* the resultant DDG of lib.
We use the terms *"imitation"* and *"potential imitation"* interchangeably for the rest of the paper.

B. Pre-processing Library APIs

For safety reasons, additional code, such as $null$ reference checks, precondition and state assertions, and exception handlers, is often instrumented into library APIs. Although these code snippets have no or little contribution to the core functionality, they result in more vertices and edges generated in their DDGs. In practice, we observe that it is not uncommon to have a client imitating a library API yet missing out such code snippets. Consequently, the trace subsumption relation between a library API and a client may not hold although it is a valid imitation. So we present three heuristics to remove such safety-checking code from library APIs. Note that all DDGs of library APIs are constructed after these heuristics are applied to API implementation.

Null Reference Checks. There are two code patterns we target at in library APIs as follows.

```
// Null Pattern 1           // Null Pattern 2
if (a == null) {            if (a != null) {
    return a constant;          ...;
} else {                        a.X();
    ...;                        ...;
    a.X();                  } else {
    ....;                       return a constant;
}                           }
```

If an invocation to a method $X()$ on an object a is guarded with a *null* reference check on a, and if a is null the API simply returns a constant, then we remove the guard *(a == null)* or *(a != null)* in the if-else condition, and the statement returning a constant.

Assertions. The pattern below captures the situation when a property ($e.g.$ preconditions or state invariant) is violated, an exception is thrown.

```
if (...) {
    ...;
    throw new Exception(...);
} else {
    ...;
}
```

In this case, we remove the conditional with the branch which throws the exception.

Exception Handlers. One observation about exception handlers is that if an exception is caught by libraries in a try-catch statement, usually either the error is logged or the exception is re-thrown. Hence we simply discard all catch blocks in libraries.

```
try {
    ...;
} catch(...) {
    ...; // log or re-throw the exception.
}
```

C. Trace Subsumption Checking Algorithm

For ease of understanding, we divide our algorithm into two parts. This section depicts how to check trace subsumption relation of two DDGs, whereas the other Section III-D shows how trace subsumption checking is combined with method inlining to detect library API imitation.

Algorithm 1 checks trace subsumption relation. The procedure $Subsume$ takes as input two graphs G_{lib} and G_{clt} from a library and a client methods respectively, and returns $true$ if G_{clt} trace subsumes G_{lib}, false otherwise. The $stack$ in line 1 is a stack and each element is a pair $(libv, matches)$, of which $libv$ is a vertex in G_{lib} and $matches$ is a set of vertices in G_{clt} matching $libv$. The set $visited$ in line 2 stores the visiting information – a set of visited pairs – to avoid repeated checking. The algorithm performs a case analysis over the nature of vertices: entry and non-entry vertices.

1) Checking Entries: In line 3–15, the procedure iterates over the entries of G_{lib}. For each entry en, it collects all the matching vertices in G_{clt} into set $matches$ in line 5–9. If there is no match, the algorithm returns $false$, otherwise adds the pair $(en, matches)$ to $stack$.

2) Checking Non-Entries: In line 16–28, the procedure performs step-wise trace subsumption relation check. It repeats the process until $stack$ is empty or an un-matched library vertex is found. In line 17, it first pops a pair $(libv, matches)$ from $stack$, Then for each successor $succ$ of $libv$, it calls the procedure $FindMatches$ to search the matching vertices in G_{clt}, of which each is a successor of a vertex of $matches$. In line 20, if there is no match in G_{clt}, the algorithm returns $false$, otherwise, the matching pair $(succ, matches)$ is pushed into $stack$ if it has not been visited. Finally, if $stack$ becomes empty, the algorithm ends with $true$ at line 28.

Algorithm 1 *Boolean* **Subsume(**$DDG\ G_{lib}$**,** $DDG\ G_{clt}$**)**

```
Input:
$G_{lib}$: a DDG of a library method
$G_{clt}$: a DDG of a client method
Output: $true$ if $G_{clt}$ trace subsumes $G_{lib}$, $false$ otherwise
Body:
 1: $Stack\ stack := []$
    {each stack element is a pair $(lv, cs)$ where $lv$ is a vertex in $G_{lib}$, and $cs$ is a set of vertices in $G_{clt}$ matching $lv$}
 2: $Set\ visited := \emptyset$
    {$visited$ stores all pairs which are ever in $stack$}
    {Locate all the matches in $G_{clt}$ for each entry of $G_{lib}$.}
 3: for each entry $en$ of $G_{lib}$ do
 4:     $Set\ matches := \emptyset$
        {Search for matching vertices in the whole $G_{clt}$ for $en$}
 5:     for each vertex $v$ of $G_{clt}$ do
 6:         if $match(en, v)$ then
 7:             $matches := matches \cup \{v\}$
 8:         end if
 9:     end for
10:     if $matches = \emptyset$ then
11:         return $false$
12:     else
13:         $stack.push((en, matches))$
14:     end if
15: end for
    {Check trace subsuming relation in DFS style.}
16: while $stack \neq []$ do
17:     $(libv, matches) := stack.pop()$
18:     for each successor $succ$ of $libv$ in $G_{lib}$ do
19:         $Set\ matches' := FindMatches(G_{clt}, matches, succ)$
20:         if $matches' = \emptyset$ then
21:             return $false$
22:         else if $(succ, matches') \notin visited$ then
23:             $visited := visited \cup \{(succ, matches')\}$
24:             $stack.push((succ, matches'))$
25:         end if
26:     end for
27: end while
28: return $true$
```

```
Proc: $Set$ FindMatches($DDG\ G_{clt}$, $Set\ starts$, $Vertex\ libv$)
Input:
$G_{clt}$: a DDG of a client method
$starts$: a set of vertices in $G_{clt}$ as the starts of the search
$libv$: a vertex in a library DDG to match
Output: a set of vertices in $G_{clt}$ matching $libv$
Body:
 1: $Set\ result := \emptyset$
 2: for each vertex $start \in starts$ do
 3:     for each successor $succ$ of $start$ in $G_{clt}$ do
 4:         if $match(libv, succ)$ then
 5:             $result := result \cup \{succ\}$
 6:         end if
 7:     end for
 8: end for
 9: return $result$
```

D. Overall Algorithm

For illustrative purpose, we present Algorithm 2, which is a simplified version of API imitation detection algorithm. It takes as input a library and a client methods lib and clt, and returns $true$ if clt potentially imitates lib, that is, after some method calls in lib and clt are inlined, the resultant DDG of clt trace subsumes that of lib.

This algorithm assumes that each method is initially associated with a constructed DDG, which is referred to as *original* DDG. We manipulate copies of original DDGs, and all the inlining operations are conducted at DDG level, due to overhead consideration that constructing DDGs from code involving data dependency analysis is computationally expensive compared to simply cloning graphs. In line 3, all the *Call* vertices in both original client and library DDGs are collected into set *calls*. We enumerate all the subsets of *calls* in line 4, and for each subset the corresponding DDGs are inlined

Algorithm 2 *Boolean* **Detect**(*Method lib*, *Method clt*)

```
Input:
lib: a library method
clt: a client method
Output: $true$ if $clt$ potentially imitates $lib$, $false$ otherwise.
Body:
1: $DDG\ G_{lib}$ := the DDG of $lib$
2: $DDG\ G_{clt}$ := the DDG of $clt$
3: $Set\ calls$ := all Call vertices in both $G_{lib}$ and $G_{clt}$
4: for each Call vertex set $sub \in \mathcal{P}(calls)$ do
5:     $DDG\ G'_{lib}$ := clone a copy of $G_{lib}$
6:     $DDG\ G'_{clt}$ := clone a copy of $G_{clt}$
7:     for each Call vertex $call \in sub$ do
8:         $DDG\ G_{call}$ := clone a copy of DDG that $call$ refers to
9:         if $call$ is in $G_{lib}$ then
10:            inline $G_{call}$ into $G'_{lib}$
11:        else
12:            inline $G_{call}$ into $G'_{clt}$
13:        end if
14:    end for
15:    if $Subsume(G'_{lib}, G'_{clt})$ then
16:        return $true$
17:    end if
18: end for
19: return $false$
```

into G'_{clt} or G'_{lib} at line 8–13; then subsumption relation is checked for the two graphs at line 15. We repeat this process until the enumeration is finished or we find a pair of G'_{lib} and G'_{clt} such that $G'_{lib} \sqsubseteq G'_{clt}$.

E. Pruning Method Inlinings

As the runtime overhead of Algorithm 2 is exponential to the size of $calls$ in line 3, we employ several heuristics to minimize the size of $calls$, such as no inlining of recursive calls and the followings. Our experimental result shows that the heuristics are effective, making our prototype efficient even for large Java systems.

1) Only Inlining Single-Target Calls: Polymorphism is a key feature of object-oriented programming languages. A *virtual* method call[1] can be dynamically dispatched to a different method body at runtime. Thus a *Call* vertex may statically correspond to multiple DDGs. We use class hierarchy analysis to resolve the possible targets of each virtual method call, and remove it if it corresponds to multiple or no targets.

2) Feedback-Guided Inlining: Rather than enumerating all subsets of $calls$ in a deterministic order, we only inline some subsets of them based on the feedback from invocation of $Subsume$. When $Subsume$ is invoked at line 15 in Algorithm 2, in our prototype, not only do we get the result $true$ or $false$ of trace subsumption checking, but also a library vertex $unmatched$ and a set of method calls $calls_{client}$ in the client DDG. The vertex $unmatched$ stores the un-matched library vertex $libv$ if $Subsume$ returns $false$ and $libv$ is a *Call* vertex, otherwise it is $null$. In $calls_{client}$, each call corresponds to a DDG g and g has at least one vertex matching a library vertex visited in the checking. Set $calls_{client}$ will be constructed in Algorithm 1 when searching for vertices in client DDG that match the library vertex $libv$. The construction is based on the type of $libv$ vertex. We first describe how this set is constructed, and then explain how $unmatched$ and $calls_{client}$ can be used to guide selection of call subsets.

Entry. If $libv$ is an entry but not *Formal* or *This*, then $calls_{client}$ contains those method calls in the client which invoke a DDG with an entry matching $libv$. Nothing is added to $calls_{client}$ when $libv$ is either a *Formal* or a *This* vertex

Non-Entry. If $libv$ is a non-entry vertex, method calls are collected in the procedure *FindMatches* in Algorithm 1. If the vertex $succ$ at line 3 is a *Callee* (or i-th *Actual*) of a method call c, we include c in $calls_{client}$ if in the DDG of c the vertex succeeding *This* (or i-th *Formal*) matches $libv$.

The set $calls_{client}$ together with $unmatched$ constitutes a smaller set set_{small} of calls which are candidates for inlining. The procedure $Detect$ in Algorithm 2 enumerates all the subsets of set_{small}. At each enumeration, $Detect$ gets feedback from $Subsume$ and may add new method calls to set_{small}. $Detects$ terminates when there is no more new subset to enumerate or when an imitation is found.

F. Post Validation of Imitations

Given a client DDG G_{clt} and a library DDG G_{lib}, even when $Subsume(G_{lib}, G_{clt})$ returns $true$, it might still be the case that the client is not imitating the library API. We find that false positives are mostly due to the two scenarios described below, and thus post-validating the claim $G_{lib} \sqsubseteq G_{clt}$ is needed before $Subsume$ returns $true$ at line 28 of Algorithm 1.

Unmatched Inlined Vertices in Client.

```
Lib(){    Clt(){          toInline(){   Clt'() {
  a();      toInline();     a();          Lib();
  b();      b();            x();        }
}         }                 y();z();}
```

This happens when some vertices in G_{clt} come from inlined method calls, and these vertices do not match any vertex in G_{lib}. Consider the simple example above, we assume all method calls are data *independent* of one another. After we inline method *toInline* into *Clt* and construct a DDG, the new DDG trace subsumes the DDG of *Lib* with $x()$, $y()$ and $z()$ matching no library vertex. However, we cannot replace the body of *Clt* with a single invocation to *Lib* shown as method *Clt′*, as *Clt′* does not provide the behavior of methods $x()$, $y()$ and $z()$. One might suggest to append $x()$, $y()$ and $z()$ to *Clt′*, but this begs the question of whether *Clt* is trying to imitate *Lib*. To eliminate this case, we require all inlined vertices in client match at lease one library vertex.

Matching All References to Library Locals.

```
Lib(){                Clt(){
  a = compute();        a = compute();
  return a.X();         a.Y();
}                       return a.X();
                      }
```

[1]Java has four invocation instructions: *invokeStatic* for static methods; *invokeSpecial* for constructors, methods of super classes and private methods; *invokeVirtual* and *invokeInterface* for virtual methods

In the example above, the DDG of client Clt trace subsumes the DDG of library Lib. In order to refactor Clt with Lib, we require the local variable a, and its references, in Clt to become a local in Lib. This is not feasible as the original reference $a.Y()$ is not available in Lib. To eliminate this case, we require that every successor of a non-entry *Value* vertex in a client DDG matches at least one library vertex.

IV. Case Studies

We have implemented a prototype on top of the program analysis framework Wala[2]. Although the prototype currently handles Java programs, its underlying technique is applicable to object-oriented programs in general. Indeed, we simply need to add new front-end to convert programs in other object-oriented languages to data-dependency graphs.

Our prototype works on compiled bytecode. The motivations are: (1) source code of libraries is not always available, and (2) bytecode instructions are primitive – with all advanced loop structures being translated into $if \cdots goto$ structures; this translation normalizes the code and minimizes the number of vertex types in DDGs.

The system accepts a set of library class files and a set of client class files, and returns a list of imitation pairs. Each pair is composed of a library API and a client method imitating the API with imitation information. The imitation information contains the location ($i.e.$, line number and declared method) of each vertex in the identified DDGs.

We have applied this prototype to ten open-source Java projects from average sized Checkstyle[3] to large JBoss. To evaluate the detection performance, we employed the notion of *precision* defined as follows.

$$precision = \frac{number\ of\ true\ positives}{number\ of\ reported\ imitations} \tag{1}$$

A. Experimental Setup

We used the compiled code of ten diversified open-source Java projects as experiment subjects ranging from application server, server side infrastructure, object relation mapping framework to code analysis tool.

The subject information is shown in Table III. The first two columns show the project names and the version numbers. The third column shows the total number of methods excluding abstract, native and interface methods in client code.

In the case studies, for each client method clt of a project, we tested clt against a set of library APIs $libs$ in that project. As a heuristic, $libs$ does not include trivial library methods which only read or write a value (*i.e.* fields, constants, local or global variables). Based on the choice of $libs$, we conducted the following two experiments to demonstrate our technique is able to detect imitations with high precision.

The testbed is a PC with Intel Core 2 Quad CPU 3.0GHz and 8GB memory. To take full advantage of multi cores, we parallelized the prototype by dispatching detection tasks to six threads.

[2]http://wala.sourceforge.net/

[3]http://checkstyle.sourceforge.net/

TABLE III
Experimental Subject Information

Project	Version	Client Methods
JBoss	5.0.1	34744
SpringFramework	2.5.5	20490
Hibernate	3.3.1	16958
ArgoUML	0.20	8384
iReport	3.0.0	11187
JasperReports	3.1.4	10431
JasperServer	3.1.0	8147
FreeMind	0.9.0	5015
Jajuk	1.7.1	2923
CheckStylePlugin	4.4.2	1108

B. Detecting Imitation of Explicitly Imported Libraries

In this experiment, for each client method clt declared in class C, we constructed the library API set $libs$ by collecting all the visible library APIs whose classes have been explicitly imported into C. Table IV lists the detection statistics.

TABLE IV
Result of Imitation Detection for Explicitly Imported Libraries

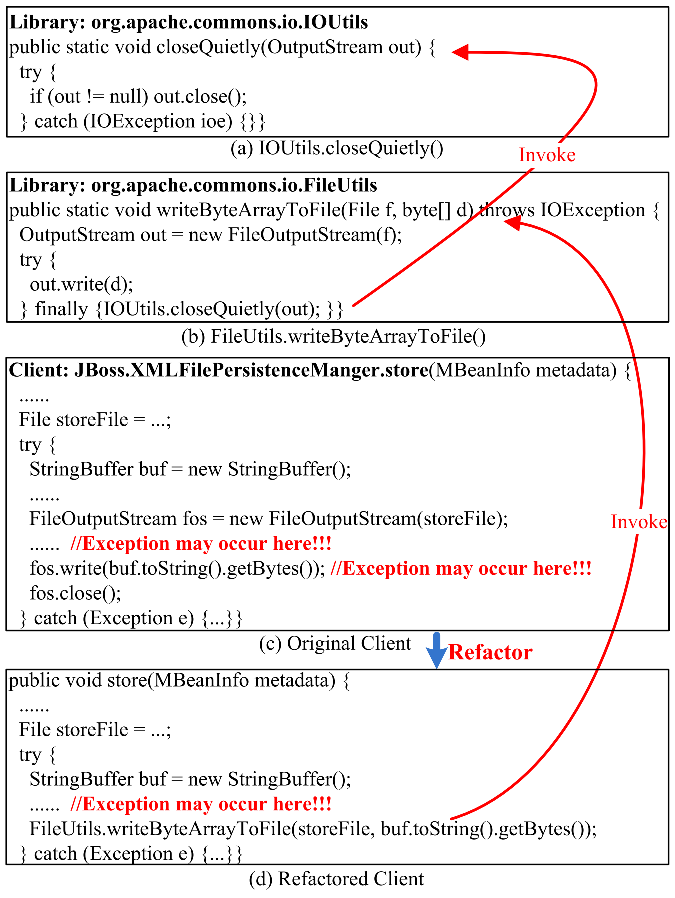
Project	Library Methods	Time (sec.)	True Posi.	All	Prec.
JBoss	683220	101	226	288	78%
SpringFramework	412931	47	1	2	50%
Hibernate	38373	23	0	1	0
ArgoUML	54721	20	43	46	93%
iReport	611224	36	8	8	100%
JasperReports	298571	24	1	1	100%
JasperServer	617429	28	29	31	94%
FreeMind	32492	12	1	1	100%
Jajuk	208048	13	2	2	100%
CheckStylePlugin	40029	10	2	3	67%
Total	2997038	314	313	383	82%

The second column is the number of library APIs in each project. The third column displays the time spent on each subject in seconds. Column *True Posi.* records the number of true positives, while column *All* is the number of reported imitation pairs. The last column is the precision computed based on columns *True Posi.* and *All*. In total, our technique reported 313 true imitations with high precision average of 82%. Although the code size of the ten projects is large, it only took 314 seconds for our prototype to finish this experiment, hence our technique is scalable and efficient.

C. Detecting Imitation of Static Libraries

Most of the static library methods encapsulate common utilities, which can be widely used. Figure 3 demonstrates how client code can be improved by a static library.

The static library method $writeByteArrayToFile()$ of class $FileUtils$ writes a byte array to a file. Internally, it opens a stream to the file, writes the array and closes the stream. It guarantees that no matter whether exceptions occur during the writing, the created stream will be finally closed. Our system detects that in client class $XMLFilePersistenceManager$ the method $store$ imitates this library API, and can be refactored to the code in Figure 3(d), which eliminates two method calls.



```
Library: org.apache.commons.io.IOUtils
public static void closeQuietly(OutputStream out) {
 try {
   if (out != null) out.close();
 } catch (IOException ioe) {}}
```

(a) IOUtils.closeQuietly()

```
Library: org.apache.commons.io.FileUtils
public static void writeByteArrayToFile(File f, byte[] d) throws IOException {
 OutputStream out = new FileOutputStream(f);
 try {
   out.write(d);
 } finally {IOUtils.closeQuietly(out); }}
```

(b) FileUtils.writeByteArrayToFile()

```
Client: JBoss.XMLFilePersistenceManger.store(MBeanInfo metadata) {
 ......
 File storeFile = ...;
 try {
   StringBuffer buf = new StringBuffer();
   ......
   FileOutputStream fos = new FileOutputStream(storeFile);
   ......  //Exception may occur here!!!
   fos.write(buf.toString().getBytes()); //Exception may occur here!!!
   fos.close();
 } catch (Exception e) {...}}
```

(c) Original Client

```
public void store(MBeanInfo metadata) {
 ......
 File storeFile = ...;
 try {
   StringBuffer buf = new StringBuffer();
   ......  //Exception may occur here!!!
   FileUtils.writeByteArrayToFile(storeFile, buf.toString().getBytes());
 } catch (Exception e) {...}}
```

(d) Refactored Client

Fig. 3. An Imitation Example of Static Library API in JBoss

After investigating into the original client, we find another benefit from calling *writeByteArrayToFile*(). The original client code subjects to resource leak. In Figure 3(c), exceptions may arise at the lines with highlighted comments. If they occur, the stream object *fos* will not be closed until it is garbage collected. In contrast, the new client has no such problem.

In this experiment, we selected a set of static library methods such as those in apache commons.io, commons.collection and commons.lang, and tested whether any of them could be applied to client methods.

TABLE V
RESULT OF IMITATION DETECTION FOR STATIC LIBRARIES

Project	**Library Methods**	**Time (sec.)**	**True Posi.**	**All**	**Prec.**
JBoss	1703	130	40	56	71%
SpringFramework	1518	61	35	49	71%
Hibernate	545	30	16	22	73%
ArgoUML	0	13	0	0	–
iReport	328	38	1	1	100%
JasperReports	118	46	1	1	100%
JasperServer	1523	36	15	17	88%
FreeMind	492	15	0	0	–
Jajuk	465	15	5	5	100%
CheckStylePlugin	1258	11	3	4	75%
Total	7950	396	116	155	75%

Table V lists the detection statistics, following the same format as Table IV. Our prototype reported 116 true imitations with precision average of 75%. It took 396 seconds to finish, more than the experiment on imported library as we need check each client method against all the static libraries.

D. Analysis

1) False Negatives: In the two experiments, the high precision is mainly benefited from the employment of the trace subsumption relation between library and client DDGs, which capture the semantic ordering of statements. However there is still 18–25% false positive rate, and we categorize the causes as follows:

- Approximation: In our matching rules, we allow approximation for *Binary*, *Const*, *etc.*
- Pre-processing: Since we remove all exception handlers in library APIs, we may over-prune their behaviors. A few library APIs continue their services even though an exception arises. For example, an API tries to dynamically load a driver of high performance, but gets a driver not found exception, then in the catch block, it loads the default drive instead. We fail to capture the behaviors in the catch block due to preprocessing.

2) Imitation Size: Our tool is able to detect examples reported in [1]. The class of imitated client code found are of sizes from 1 to 8 lines. While the imitation size is small, the example depicted Figure 1 shows that such imitation can be tricky to detect.

V. Discussion

This section discusses the rationale behind the choice of data dependency graphs as the basic data structure, and then presents major threats to validity.

A. Why Data Dependency Graphs?

Program dependency graph (PDG) [2] is a comprehensive semantic representation of programs. It has two edge types, control dependency edge and data dependency edge. A PDG with data dependency edges pruned is referred to as a Control Dependency Graph (CDG). Similarly, a PDG with control dependency edges pruned is referred to as a Data Dependency Graph. In this paper, we only employ DDGs as we believe that data flows among program statements provide enough semantic information for imitation detection. Our experiment results provide strong support to our hypothesis.

Employing only CDGs in representing programs does not work well as it carries little semantic information. A control dependency edge only connects a conditional statement (*e.g.* for, while) to a statement whose execution depends on the value of the conditional. If a method body contains no conditional statements, then its CDG is actually a set of vertices with no edges. The study in [1] has shown that the technique based on individual element matching without considering element ordering requires additional heuristics to maintain reasonable precision.

Considering PDGs for program representation on the other hand can improve the precision, yet we will risk missing valid imitations, as programmers may write imitated client code with different control flow structures from that of libraries.. For instance we will lose the following interesting imitation if we consider both data and control dependencies.

```
Library: Tigris GEF
public int FigText.getMinimumHeight() {
l1: if (_fm != null)
l2:     return _fm.getHeight();
l3: if (_font != null)
l4:     return _font.getSize();
l5: return 0;
}

Client: ArgoUML
Dimension FigSignleLineText.getMinimumSize() {
    Dimension d = new Dimension();
    int maxH = 0;
    Font font = getFont();
c1: if (font == null)
c2:     return d; //return empty dimension (0,0)
    ......
c3: if (getFontMetrics() == null) {
c4:     maxH = font.getSize();
    } else {
c5:     maxH = getFontMatrics().getHeight();
c6: }
    ......
}
```

The class $FigSignleLineText$ is a subclass of $FigText$, the method $getFontMetrics()$ is a getter method of the field $FigText.\_fm$, and $getFont()$ is a getter method of the field $FigText.\_font$. The client imitates the library behavior and the code from line c_3 to c_6 can be replaced by

```
    maxH = this.getMinimumHeight();
```

The PDGs of the two methods have different control dependency edges. The library PDG has control dependency edges $l_1 \rightarrow l_2$ and $l_3 \rightarrow l_4$, whereas the client PDG has $c_1 \rightarrow c_2$, $c_3 \rightarrow c_4$ and $c_3 \rightarrow c_5$. These control dependency edges will hinder the detection of this imitation pair, as for the trace $l_3 \rightarrow l_4$ in library PDG, there is no trace in client PDG that subsumes it.

B. Threats to Validity

Similar to other empirical studies, there is a threat to validity in interpreting the results. Specifically, the classification of the validity of imitated reports has so far remained manual. To help ensure the quality of the imitation classification process, each reported imitation was independently classified by two evaluators besides the programmer. Consequently, although our evaluation criterion is that the reported library should make client code shorter and cleaner, they are still subject to many ways of interpretation.

VI. Related Work

One pioneer study on detecting imitations of library APIs is that of Kawrykow and Robillard [1], which is also closest to our work. They propose a static analysis based approach to detecting the item-set similarity between clients and libraries. It first abstracts a set of program elements (including fully-qualified field or method signatures and referenced types) from each method body, and then tests whether each element in the library element set can be matched against any one in the client set. A library element can match a client element in a direct, indirect or nested fashion. The indirect and nested styles are designed to handle complex imitations like the motivating example in Figure 1. If all the library elements can be matched, then the library is said to be possibly imitated by the client. For the same ten open-source Java projects as we use, their approach reports 405 valid imitations in total with precision average of 31% for explicitly imported libraries.

As shown in Section IV, our approach reports 313 valid imitations with much higher average precision 82%. (We are not able to compare two of the experimented results in detail as the data is not available.) The high precision stems from two sources: our approach is semantics-based, and the trace subsumption relation employed here can capture the data flow semantics inside method bodies; second, we use two heuristics to post-validate detected imitations and eliminate invalid ones. Furthermore, we also conduct case study on imitations of static libraries encapsulating common functionalities, and detect 116 true imitations with precision average of 74%. Nevertheless, there is no free lunch. We fail to identify a few valid imitations, and as we construct data dependency graphs and test trace subsumption relations, the overhead of our approach, albeit manageable, is higher than theirs.

Another closely related work is Krinke's approach [4]. Given two program dependency graphs, Krinke tries to construct maximal similar subgraphs representing similar code. He first chooses a starting vertex in each PDG and extends isomorphic paths from it in a step-wise fashion until the paths cannot be grown. Finally the two sets of paths, each from a PDG, are considered as maximal subgraphs similar to each other. From a technical viewpoint, his approach is quite similar to ours. That is, we both employ path/trace-based algorithms to measure the similarity between PDGs/DDGs. However there are several differences: technically, he needs to first specify starting vertices to grow maximal similar graphs; second he does not consider complex similar code involving method inlining; from the viewpoint of motivation, he tries to identify similar code whereas we aim to improve usages of library APIs.

Two other related research directions are duplicate code and plagiarism detection. Gabel *et al.* proposes a scalable technique to detect semantic clones in C/C++ programs [2]. They decompose program dependency graphs into semantic threads and construct abstract syntax trees from threads. Then each tree is transformed into a feature vector. Lastly they use locality sensitive hashing [5] to cluster the nearest vectors and regard each cluster as a clone group. As they reduce the graph similarity to a nearest vector problem, their approach is fairly scalable and has been applied to large programs. Komondoor and Horwitz [6] [7] use program dependency graphs to find semantically identical code fragments and automatically extract procedures from these fragments. Since their approach is based on subgraph isomorphism which is NP-complete, it cannot scale well. Liu *et al.* detect software plagiarism by checking γ-isomorphism of program dependency graphs [8]. A statistical lossy filter is used to to prune the plagiarism search space for scalability reason. Schuler *et al.* use object-level call sequences as an API birthmark based on the observation of program interaction at runtime, which is resilient to obfuscation techniques [9].

Another relevant work is API usage mining and adaptation. Nguyen *et al.* propose a graph-based object usage model (groum) similar to a program dependency graph to represent multiple object usage patterns [10]. They first construct a groum for each method and mine frequent sub-groums from a set of groums as usage patterns. They also propose an approach to recommending API usage adaptation for evolving libraries [11]. Given two versions of a library, they first extract the library API usage skeletons from client code before and after library migration, next compare the skeletons to recover API usage adaptation patterns, and finally suggest edit operations for clients to be updated to the new version of the library. They focus on adapting existing usage of a library API to that of its new version, whereas we try to eliminate imitations found at client site of library APIs. Xnippet [12] is a context-sensitive code assistant framework that allows developers to query a sample repository for code snippets that are relevant to the programming task at hand. MAPO [13] and Prospector [14] both accept code queries, perform data mining at background and return frequent API usage patterns. Researchers also have conducted studies on mining API usage patterns via dynamic analysis. Lo *et al.* propose an iterative pattern mining algorithm to discover temporal specification of method calls from execution traces [15]. Pradel *et al.* mine object usage specifications in the form of finite automata from large method traces [16].

Our trace subsumption relation is inspired by trace refinement [17]. Trace refinement states that a program P_1 refines another program P_2 whenever the set of execution traces of P_1 is a subset of that of P_2. It ensures that P_1 cannot have any behavior which is not permitted by P_2. Thus, trace refinement is suitable for describing how certain properties of a system must be preserved between two programs, particularly in our case, the common data flows.

VII. Conclusion & Future Work

Imitating API codes represents an ineffective usage of libraries as such re-implementation is not necessary and the existence of imitated codes creates maintenance burden. In this paper, we propose a graph-based approach to detecting such imitations. Our technique utilizes *trace subsumption* relation of data dependency graphs to characterize the similarity between client code and the imitated library. We have built a prototype and investigated its utility on ten sizable open-source projects. The experiment shows that our approach can report 313 valid imitations in total with high precision average of 82% for explicitly imported library APIs, and 116 valid imitations with precision average of 75% for static library APIs.

In the future, we would like to explore the feasibility of path-sensitive analysis to further improve our current solution in terms of precision and number of detected valid imitations.

Acknowledgment

We are grateful to David Kawrykow at McGill University for sharing the detail of iMaus, and to the reviewers for their valuable comments. This work is partially supported by a research grant R-252-000-403-112.

References

[1] D. Kawrykow and M. P. Robillard, "Improving API Usage through Automatic Detection of Redundant Code," in *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 111–122.

[2] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[3] K. J. Ottenstein, "Data-flow Graphs as an Intermediate Program Form." Ph.D. dissertation, Purdue University, 1978.

[4] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," in *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering*, 2001, p. 301.

[5] A. Gionis, P. Indyk, and R. Motwani, "Similarity Search in High Dimensions via Hashing," in *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, 1999, pp. 518–529.

[6] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," in *SAS '01: Proceedings of the 8th International Symposium on Static Analysis*, 2001, pp. 40–56.

[7] ——, "Semantics-preserving Procedure Extraction," in *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2000, pp. 155–169.

[8] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis," in *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006, pp. 872–881.

[9] D. Schuler, V. Dallmeier, and C. Lindig, "A Dynamic Birthmark for Java," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ser. ASE '07, 2007, pp. 274–283.

[10] 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: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 383–392.

[11] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '10, 2010, pp. 302–321.

[12] N. Sahavechaphan and K. Claypool, "XSnippet: Mining for Sample Code," *SIGPLAN Not.*, vol. 41, no. 10, pp. 413–430, 2006.

[13] T. Xie and J. Pei, "MAPO: Mining API Usages from Open Source Repositories," in *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, 2006, pp. 54–57.

[14] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid Mining: Helping to Navigate the API Jungle," *SIGPLAN Not.*, vol. 40, no. 6, pp. 48–61, 2005.

[15] D. Lo, S.-C. Khoo, and C. Liu, "Efficient Mining of Iterative Patterns for Software Specification Discovery," in *KDD '07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2007, pp. 460–469.

[16] M. Pradel and T. R. Gross, "Automatic Generation of Object Usage Specifications from Large Method Traces," in *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 371–382.

[17] A. W. Roscoe, C. A. R. Hoare, and R. Bird, *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.