A Graph-based Approach to API Usage Adaptation

Hoan Anh Nguyen,[1a] Tung Thanh Nguyen,[1b] Gary Wilson Jr.,[2c] Anh Tuan Nguyen,[1d]
Miryung Kim,[2e] Tien N. Nguyen[1f]

Iowa State University[1] and The University of Texas at Austin[2]

{hoan[a],tung[b],anhnt[d],tien[f]}@iastate.edu, gwilson@austin.utexas.edu[c], miryung@ece.utexas.edu[e]

Abstract

Reusing existing library components is essential for reducing the cost of software development and maintenance. When library components evolve to accommodate new feature requests, to fix bugs, or to meet new standards, the clients of software libraries often need to make corresponding changes to correctly use the updated libraries. Existing API usage adaptation techniques support simple adaptation such as replacing the target of calls to a deprecated API, however, cannot handle complex adaptations such as creating a new object to be passed to a different API method, or adding an exception handling logic that surrounds the updated API method calls.

This paper presents LIBSYNC that guides developers in adapting API usage code by learning complex API usage adaptation patterns from other clients that already migrated to a new library version (and also from the API usages within the library's test code). LIBSYNC uses several graph-based techniques (1) to identify changes to API declarations by comparing two library versions, (2) to extract associated API usage skeletons before and after library migration, and (3) to compare the extracted API usage skeletons to recover API usage adaptation patterns. Using the learned adaptation patterns, LIBSYNC recommends the locations and edit operations for adapting API usages. The evaluation of LIBSYNC on real-world software systems shows that it is highly correct and useful with a precision of 100% and a recall of 91%.

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

General Terms Algorithm, Design, Experimentation, Reliability

Keywords Software Evolution, API Usage Model, API Evolution, API Usage Adaptation, Program Differencing

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

1. Introduction

Reusing existing software components by accessing their implementations through their Application Programming Interfaces (APIs) can reduce the cost of software development and maintenance. When libraries provide their functionality through public interfaces (e.g., types, methods, and fields in Java), clients are expected to respect the contract assumed by the libraries by using the correct names of the APIs, passing the right arguments, following the intended temporal orders of API invocations, etc.

When library components evolve to accommodate new feature requests, to fix bugs, and to meet new standards, changes in API declarations in libraries could cause existing clients to break. For example, when an API signature modification requires more input parameters or a different return type, clients need to pass additional input arguments or to process a returned object differently.

Existing analysis techniques that can be used for adapting API usage code in client applications have the following limitations. First, existing research techniques such as CatchUp! [15] and MolhadoRef [11] require library maintainers and client application developers to use the same development environment to record and replay refactorings. Other techniques require library developers to manually write expected adaptations in client code as rules [7]. Second, existing API usage modeling and extraction techniques [1,12,37,39] are limited by simplified representations such as a sequence of method calls. Thus, they cannot capture the complex control and data dependencies surrounding the use of APIs. For example, SemDiff [8] models API usages in terms of method calls, so it can support changing the target of calls to modified APIs but cannot add the control structure that surrounds the calls to a new replacement API.

Hypothesizing that changes to API usage caused by the evolution of library components may involve complex changes, we developed a set of graph-based models and algorithms that can capture updates in evolving libraries and updates in client applications associated with changes in the libraries, and an algorithm that generalizes common edit operations from a set of API usage code fragments before and after library migration.

Our API usage code adaptation framework takes as input the current version of a client application, both the old version and the new version of a library under focus, and a set of programs that already migrated to the new library version. Our framework consists of four main components: (1) an ORIGIN ANALYSIS TOOL (OAT) that maps corresponding code elements between two versions, (2) a CLIENT API USAGE EXTRACTOR (CUE) that extracts API usage skeletons from client code and the use of APIs within the library, (3) an API USAGE ADAPTATION MINER (SAM) that automatically infers adaptation patterns from a set of API usage skeletons before and after the migration from the old library version to the new library version, and (4) LIBSYNC that recommends which API usage code needs to be adapted and how those code fragments need to be updated by locating API usage fragments in the client that need to be adapted and suggesting edit operations required for adaptation.

OAT models a Java program as a project tree, in which nodes represent code elements (packages, classes, interfaces, method-headers, fields, and method-bodies). It uses a tree-based alignment and differencing algorithm to map code elements and detect addition, deletion, renaming, moving, and modification of those elements.

CUE extracts the skeleton of API usage code in client systems and the test code of the APIs. We make the assumption that client systems use library components by accessing their APIs via **invocation**—directly calling API methods or instantiating objects from API classes—and via **inheritance**—declaring classes in the client by subtyping API classes. We call those two ways of using APIs as API *i-usage* (invocation) and *API x-usage* (extension). We also use the term *API usage* to refer to both types of API usages.

In particular, CUE extends Nguyen *et al.*'s graph-based object usage model (GROUM) [25] that represents both control and data dependencies among method calls and field accesses. In order to capture API usage, we extended GROUM to explicitly note the usages of external APIs via invocation and extension by modeling the types of objects passed to APIs as arguments and by modeling overriding and inheritance relationships between client methods and methods provided by external API types. In particular, CUE represents API i-usages by an invocation-based, graph-based object usage model, iGROUM, in which, *action nodes* represent method invocations, *control nodes* represent surrounding control structures, and *data nodes* represent use of types provided by external libraries. The edges represent dependencies between the nodes, for example, usage orders, input and output relations. CUE represents API x-usage by another graph-based model called xGROUM, in which each node represents a method-header and two kinds of edges represent overriding and inheritance relationships between a method-header in the client and a method-header in the library.

SAM uses an approximate graph alignment and differencing algorithm to map nodes between two usage models based on their similarity of labels and neighborhood structures, and calculates the editing operations based on the alignment. For example, the aligned nodes having different attributes are considered as replaced or updated, while unaligned nodes are considered as deleted or added. Since API usage changes are detected as change sets of editing operations, SAM mines the frequent subsets of such change sets using a frequent item set mining algorithm [2] and considers them as *API usage adaptation patterns*.

LIBSYNC has a knowledge base of API usage adaptation patterns for each library version. Given a client system and the desired version of library to migrate to, LIBSYNC identifies the locations of API usages in the client system that are associated with changed APIs. It then matches each usage with the best matched API usage pattern in its knowledge base and derives the edit operations for adapting API usages.

We have conducted an empirical evaluation of LIBSYNC on three large open-source subject systems, each uses up to 300 libraries. We have done several experiments to evaluate the correctness and usefulness of our tool in two usage scenarios: (a) API usage adaptation in different locations within a client program, and (b) adaptation in different branches of a client program (e.g. back-porting). The evaluation shows that OAT and CUE detect changes to API declarations and API usages with high accuracy, and LIBSYNC provides useful recommendation in most cases, even when API usage adaptation involves complex changes to the control logic surrounding API usages.

The key contributions of the paper include:

1. OAT, a tree-based origin analysis technique to automatically identify corresponding APIs between two versions of a library and find corresponding usage code fragments in client systems.
2. CUE, a graph-based representation that models the *context* of API usages by capturing control and data dependencies surrounding API usages; in particular, it supports API usages via method *invocations* and *subtyping* of the API types provided by an external library under focus.
3. SAM, a graph alignment algorithm that identifies API usage changes in client applications and an API usage adaptation pattern mining algorithm that generalizes common edit operations from multiple API usage skeletons before and after migration.
4. LIBSYNC, a tool that takes as input a client system and a given library version to be migrated to, and recommends the locations and potential edit operations for adapting API usage code in the client.
5. An empirical evaluation for the correctness and usefulness of LIBSYNC in adapting API usage code.

Section 2 presents motivating examples that require complex API usage adaptation in client applications that are caused by updates to libraries. Sections 3, 4, 5, and 6 detail individ-

```
1 | XYSeries set = new XYSeries(attribute,false, [ false ]);
2 | for (int i = 0; i < data.size(); i++)
3 |   set.add(new Integer(i), (Number)data.get(i));
4 | DefaultTableXYDataset dataset = new DefaultTableXYDataset(~~set~~ [ false ]);
5 | [ dataset.addSeries(set) ];
6 | JFreeChart chart = ChartFactory.createXYLineChart(..., dataset,...);
```

Figure 1. API usage adaptation in JBoss caused by the evolution of JFreeChart

ual models and algorithms that we have developed to build LIBSYNC. Section 7 shows the empirical evaluation of LIBSYNC. Section 8 describes related work and Section 9 summarizes this paper's contributions.

2. Motivating Examples

JBoss [31] is a large project that has been developed more than 6 years ago, with 47 releases. It has about 40,000 methods and uses up to 262 different libraries. Using the Subversion [32] and code search functionality in Eclipse, we manually scanned the version history of JBoss and the external libraries used by JBoss. We examined more than 200 methods that changed due to the modification of external APIs based on associated documentation, change logs, and bug reports. This section presents API usage adaptation examples that motivate our approach.

2.1 Examples of API usage via method invocations

Figure 1 illustrates an API usage adaptation example in JBoss with respect to the use of JFreeChart library. The code changes are represented with [added code] and ~~deleted code~~. The changes from JBoss version 3.2.7 to 3.2.8 were due to the modification of external APIs, XYSeries and DefaultTableXYDataset, in JFreeChart from version 0.9.15 to 0.9.17. To enable a new *auto-sorting* feature, the XYSeries constructor with two input arguments is *deprecated* and a new constructor with three input arguments is provided instead. The DefaultTableXYDataset constructor that accepts XYSeries as an input is also *deprecated*. The new constructor accepts a boolean input to activate the new *auto-pruning* feature for data points in DefaultTableXYDataset. It implies that the XYSeries object must be added *after* the initialization of the DefaultTableXYDataset object. Thus, the parameter set is replaced by a value false, and a call to DefaultTable XYDataset.addSeries is added. This example illustrates the following:

1. JBoss uses JFreeChart via creating objects from API classes (e.g. XYSeries, DefaultTableXYDataset) and calling API methods (e.g. DefaultTableXYDataset.addSeries, ChartFactory.createXYLineChart). Since an object instantiation is represented as a constructor call to an external API type, we consider this type of API usage as an usage via invocation.

2. API usage must follow specific protocols due to the dependencies between API elements. For example, a DefaultTableXYDataset object needs to be created before any XYSeries object could be added to the set. A chart needs to be created with an object of Dataset.

```
SnmpPeer peer=new SnmpPeer(this.address
                    [ ,this.port, this.localAddress, this.localPort ]);
~~peer.setPort(this.port);~~
~~peer.setServerPort(this.localPort);~~
```

Figure 2. API usage adaptation in JBoss caused by the evolution of OpenNMS

3. As API evolves, such usage protocols could change, requiring corresponding API usage adaptations. For example, the calls to deprecated methods are replaced with newly provided ones, or new method calls are added, etc. In this example, the following edit operations occurred in the client code: replacement (e.g. the constructor of XYSeries), addition (e.g. DefaultTable XYDataset.addSeries), and update of input/output dependencies (e.g. the object XYSeries no longer immediately depends on DefaultTableXYDataset.<init> but instead on DefaultTableXYDataset.addSeries).

This example shows that existing state-of-the-art adaptation approaches (e.g. SemDiff [8], CatchUp [15]) could not support the API usage adaption because they assume that the adaptation needed in client code is simply individual method-call replacements or type declarations. They do not consider the context of API usages, the dependencies between method calls, and the differences between the extension and invocation of API methods.

Figure 2 shows another API usage adaptation example. From version 1.6.10 to 1.7.10, in the OpenNMS library, a new constructor with four parameters is added for initializing SnmpPeer. Such API change requires adding a call to the new constructor and the removal of two subsequent calls to setter methods as in Figure 2. This adaptation from JBoss version 3.2.5 to 3.2.6, although simple in meaning, is complex in term of edit operations: it involves one constructor-call replacement and two method-call deletions. Importantly, all edited calls are dependent. SemDiff [8] could suggest the replacement of the old constructor call, however, it does not suggest the setter method-call deletions because it does not consider the usage context when recommending adaptations.

2.2 Examples of API usage via inheritance

Figure 3 shows an API usage example via inheritance. The API class Serializer in the org.apache.axis.encoding package provides the writeSchema method. The AttributeSerializer class in JBoss inherits the Serializer class and overrides the writeSchema method. When the input signature of the writeSchema is changed by requiring a Class type argument and returning Element instead of boolean, the signature of the overriding method needs to be updated accordingly to properly override the writeSchema method.

Figure 4 shows another example. Class C=EJBProvider in JBoss inherits the class with the same name A=EJBProvider in

Change in Apache Axis API

```
package org.apache.axis.encoding;
class Serializer ...{
 public abstract ~~boolean~~ Element writeSchema( Class c, Types t)...
 ...
```

Change in JBoss

```
package org.jboss.net.jmx.adaptor;
class AttributeSerializer extends Serializer {
 public ~~boolean~~ Element writeSchema( Class clazz, Types types)...
 ...

class ObjectNameSerializer extends Serializer {
 public ~~boolean~~ Element writeSchema( Class clazz, Types types)...
 ...
```

Figure 3. API usage adaptation in JBoss caused by the evolution of Axis

Change in Apache Axis API

```
package org.apache.axis.providers.java;
class EJBProvider ... {
 protected Object ~~getNewServiceObject~~ makeNewServiceObject (...)
 ...
```

Change in JBoss

```
package org.jboss.net.axis.server;
class EJBProvider extends org.apache.axis.providers.java.EJBProvider {
 protected Object ~~getNewServiceObject~~ makeNewServiceObject (...)
 ...
```

Figure 4. API usage adaptation in JBoss caused by the evolution of Axis

the Apache Axis library. The method method m=getNewService-Object of A was renamed into makeNewServiceObject. Thus, its overriding method $C.m$ is also renamed accordingly.

2.3 Observations

We make the following observations based on API usage adaptation examples. First, in object-oriented programming (OOP), there are two common ways to use the API functionality (1) via **method invocation**, i.e. directly calling to API methods or creating objects of API classes; and (2) via **inheritance**, i.e. declaring classes in client code that inherit from the API classes and override their methods. Second, to use APIs correctly, client code must follow specific order of method calls or override certain methods. Thus, API usage model and API usage adaptation model must capture complex context surrounding API usages: (1) data and ordering dependencies among API usages, (2) control structures around API usages, and (3) the interaction among multiple objects of different types.

Those observations imply the necessity of a recommendation tool that helps developers in API usage adaptation to cope with evolving libraries. The tool should provide recommendations regarding where and how to do API usage adaptation. That is, given a client program using libraries and the changes to external API declarations, the tool should identify the locations to update and suggest adaptations.

3. Origin Analysis Tool (OAT)

This section discusses the origin analysis technique we have developed to map corresponding code elements (packages, classes, and methods) between two program versions. This technique is used for two different purposes: to identify modification to API declarations between two versions of a library and to map corresponding API usage code fragments between two versions of a client. This origin analysis works at or above the level of method-headers since API usages via both invocation and inheritance only refer to classes and methods. OAT views a program P (either a library or client) as a project tree $T(P)$, where each node represents a package, class, interface, or method. Each node has the following set of attributes:

- Declaration (declare(u)): For a package node, it is a fully qualified name. For a class node, it is a simple name followed by the names of the classes and interfaces that the node extends or implements. For a method node, it is a simple name, a list of parameter types, all modifiers, a set of exceptions, a return type, and associated annotations such as deprecated.
- Parent (parent(u)): It refers a node's container element.
- Content (content(u)): It represents a set of descendant nodes. For a method node, it represents the body of the method. When the source code is available, it represents the abstract syntax tree of the method body. Otherwise, it represents a sequence of byte code instructions extracted from a jar file.

For a client system P, OAT views each used library L as a sub-system, and represents L as a project tree $T(L)$. Thus, each client program is represented as a forest of several project trees. Figure 8 shows an example of a project tree of org.apache.axis.providers. Due to space limit, we show only a subset of methods of EJBProvider with some of their attributes: name, parameter types, and visibility (red square for private, yellow rhombus for protected and green circle for public).

Section 3.1 describes the types of transformations that OAT supports, Section 3.2 describes similarity measures that were used to derive one-to-one mapping between tree nodes, and Section 3.3 describes our tree alignment algorithm that maps tree nodes and derives the tree transformations from the alignment result.

3.1 Transformation Types

Suppose that two versions P_i and P_j of a program P are represented as two trees $T(P_i)$ and $T(P_j)$. The changes between 2 versions are represented as the following types of transformations from one tree to another: add(u), delete(u),

move(u)—changes to the location of node u, and update(u)-changes to u's attributes. An updated node could also be moved. For a class, an update can be performed on its name, its superclass, or its interfaces. For a method, the update can be change to its name, return type, visibility modifiers, or input signature. Those types of transformations are derived from an alignment result by considering unmapped nodes as added or deleted, and mapped nodes as moved or updated.

Figure 8 shows an example of changes found in the Axis library. Neither packages nor classes were deleted or added. Under the EJBProvider class, its method getContext was added, its method getNewServiceObject was renamed to makeNewServiceObject, the input signature of its methods getServiceClass and getEJBHome was changed to take an additional SOAP type argument, and its method getStrOption changed its visibility from private to protected.

3.2 Similarity Measures

The similarity score between two nodes is computed by summing up their declaration attribute similarity, s_d, and their content attribute similarity s_c, which are defined differently for each type of nodes.

Method Level Similarity. s_d is computed based on weighted sum of textual similarities of return types, method names, and a list of parameter types:

$s_d(u, u')$ = 0.25 * $strSim$(returntype, returntype')
+ 0.5 * $seqSim$(name, name')
+ 0.25 * $seqSim$(parameters, parameters')

in which $seqSim$ computes the similarity between two word sequences using the longest common subsequence algorithm [16], and $strSim$ computes a token-level similarity between two strings by using an idea from our prior work (Kim *et al.* [18]'s API matching). For example, given the two methods, getNewServiceObject(Context,String) and makeNewServiceObject(-SOAP,Context,String), s_d is 0.875.

$s_d(u, u')$ = 0.25 * $strSim$(Object,Object)
+ .5 * $seqSim$(getNewServiceObject,makeNewServiceObject)
+ .25 * $seqSim$([Context,String],[Context,String])
= 0.25 * (1/1) + 0.5 * (3/4) + 0.25 * (2/2) = 0.875

If the content is represented as an AST, s_c is computed by extracting a characteristic vector $v(u)$ from a method u using our prior work Exas [24]. Exas is a method to approximate the structural information of labeled trees and graphs by vectors and to measure the similarity of such trees and graphs via vector distance. If the content consists of byte code instructions, its characteristic vector $v(u)$ is an occurrence-counting vector of all the opcodes. Then, the similarity between 2 methods u and u' is computed as follows:

$$s_c(u, u') = \frac{2 * ||Common(v(u), v(u'))||_1}{||v(u)||_1 + ||v(u')||_1}$$

in which $v(u)$ a vector representation of the method content, and $Common(V, V')$ is defined as $Common(u, v)[i] = min(u[i], v[i])$. This formula is the ratio of the common part over their average size to measure the similarity.

Class and Package Level Similarity. The declaration similarity s_d is defined similarly to that for methods. The content similarity s_c is computed based on how many of their children can be mapped.

$$s_c(C, C') = \frac{2 * |MaxMatch(content(C), content(C'), sim)|}{|content(C)| + |content(C')|}$$

The $MaxMatch$ function takes two sets of entities C and C' and returns a set of pairs such that $sim(u, u')$ is greater than a chosen threshold and there exists no u'' such that $sim(u, u'') > sim(u, u')$.

3.3 Mapping Algorithm

OAT takes two project trees as input, aligns them, and computes the transformations from one tree to another. It maps nodes in a top-down order, mapping parent nodes before their children. When method m is class C's child and C' is mapped to C, we first try to map m to C''s child. If a match is not found, we assume that m is moved to another class. We adopted this strategy from UMLDiff [41] to reduce the number of candidate matches that need to be examined.

At any time, each node is placed in one of three sets: (1) AM, it is already mapped to another node, (2) PM, its parent node is mapped but it is not mapped to any node, and (3) UM, the node and its parent are not mapped. OAT maps nodes in UM first and the children of mapped ones are put in PM for further consideration. For example, when a package is mapped, its sub-packages are put in PM. The mapped ones are moved to AM, and the remaining ones that were not mapped to their parent's children are put back to UM for later processing.

When there are a large number of unmapped nodes, a pairwise comparison of all nodes in UM would be inefficient. To overcome this problem, OAT uses the following hash-based optimizations: OAT first hashes the nodes in UM by their name and only compares the nodes with the same name to find the mapped nodes in UM. For the remaining nodes in UM, it then hashes those nodes in each set by their structural characteristic vectors using the Locality Sensitive Hashing scheme (LSH) [4]. This LSH-based filtering helps OAT divide the remaining nodes in UM into the subsets with the same hashcode, and apply the $MaxMatch$ function on the nodes in each subset.

The characteristic vector of a class is summed-up from the normalized vectors of their methods. We normalize methods' vectors to have the same length of 1 before summing them up to build the vector of the containing class to avoid the problem of unbalanced sizes between those methods. However, in other cases for comparing methods, their corresponding vectors will not be normalized.

When mapping method nodes in UM, in addition to subdividing the nodes using their hash values, we also use s_d to quickly identify methods with a similar declaration. Figure 6 summarizes our origin analysis algorithm.

```
1  function MaxMatch(C, C′, sim) // find maximum weighted match
2   L = ∅
3   for (u, u′) ∈ C × C′
4    if(sim(u, u′) ≥ δ)
5     L = L ∪ (u, u′)
6   sortDescendingly(L)
7   while(L ≠ ∅)
8    (u, u′) = L.top()
9    M = M ∪ (u, u′)
10   for (v, v′) : (v, u′) ∈ L ∨ (u, v′) ∈ L
11    L.remove((v, v′))
```

Figure 5. Greedy Matching Algorithm

Figure 8 illustrates a matching process between two project trees. Two package nodes org.apache.axis.providers.java and org.apache.axis.providers.java are mapped first, and OAT then maps their class nodes. When EJBProvider classes are mapped, OAT maps their method nodes such as getNewService and makeNewService based on their declaration and content similarity.

4. Client API Usage Extractor (CUE)

This section describes CUE, a client API usage extractor. Section 4.1 presents the model and extraction algorithm for API usages via invocation. Section 4.2 presents the model and extraction algorithm for API usages via inheritance.

4.1 API Usage via Invocation

This section presents our graph-based representation for API usages via invocation and the corresponding extraction.

4.1.1 i-Usage Model

An API provides the functionality via its elements. Those elements provide the *computation* (via methods) or the storage of *data* (via objects). To use a function provided by an API, a client could call the computational elements (e.g. invoking a method) or process the data elements (e.g. initializing an object, using it as an input/output parameter). When multiple API methods/objects are used, the relations, e.g. the orders and dependencies, among those elements are important because they must follow the intended API usage specifications. Such usages are often related to the control structures (e.g. if, while) due to the branching or repetition of the computation and data processing.

CUE represents the API i-usages in clients via a graph-based model called **iGROUM** (invocation-based, GRaph-based Object Usage Model). In general, each usage is represented by a labeled, directed, acyclic graph, in which, the usages of API elements are represented as nodes, while the dependencies are modeled by edges. An *action node* represents a method invocation (i.e. a usage of an API computation element). A *data node* represents an object (i.e. a usage of an API data element). The label of each node is the fully qualified name and the signature of the corresponding method or class. An edge from an action node to another node represents the *control* and *data* dependencies. An edge from a data node to an action node shows that the corresponding object is used as an *input* of the corresponding call. Similarly, an edge with the opposite direction shows an *output* relation. Action nodes have some attributes to represent their input signature (e.g. a list of parameter types, modifiers, a return type, and exceptions that could be thrown).

```
1  function Map(T, T′) // find mapped nodes and change operations
2   UM.addAll(T, T′)
3   for packages p ∈ T, p′ ∈ T′ // map on exact location
4    if location of u and u′ is identical then Map(p, p′)
5   for packages p ∈ T ∩ UM, p′ ∈ T′ ∩ UM // unmapped pkgs
6    if Sim(p, p′) ≥ δ then SetMap(p, p′) // map on similarity
7   for each mapped pairs of packages (p, p′) ∈ M
8    MapSets(Children(p), Children(p′))) // map parent−mapped
         classes
9   for classes C ∈ T ∩ UM, C′ ∈ T′ ∩ UM // unmapped classes
10   if (C and C′ are in a text−based/LSH−based filtered subset
11   and sim(C, C′) ≥ δ) then SetMap(C, C′) // map on similarity
12  for each mapped pairs of classes (C, C′) ∈ M
13   MapSets(Children(C), Children(C′))) // parent−mapped meths
14  for methods m ∈ T ∩ UM, m′ ∈ T′ ∩ UM // unmapped meths
15   if (m and m′ are in a text−based or LSH−based filtered subset
16   and sim(m, m′) ≥ δ and dsim(m, m′) ≥ μ then
17    SetMap(m, m′) // map on similarity
18  Op = ChangeOperation(M)
19  return M, Op
20
21 function SetMap(u, u′) // map two nodes
22  M.add((u, u′))
23  UM.remove(u, u′)
24  PM.add(content(u), content(u′))
25
26 function MapSets(S, S′) // map two sets of nodes
27  M2 = MaxMatch(S, S′, sim) // use greedy matching
28  for each (u, u′) ∈ M2
29   SetMap(u, u′)
```

Figure 6. Tree-based Origin Analysis Algorithm

DEFINITION 1 (iGROUM). *An invocation-based, graph-based object usage model is a directed, labeled, acyclic graph in which:*

1. Each action node represents a method call;

2. Each data node represents a variable;

3. Each control node represents the branching point of a control structure (e.g. if, for, while, switch);

4. An edge connecting two nodes x and y represents the control and data dependencies between x and y; and

5. The label of an action, data, control, and operator node is the name, data type, or expression of the corresponding method, variable, control structure, or operator, along with the type of the corresponding node.

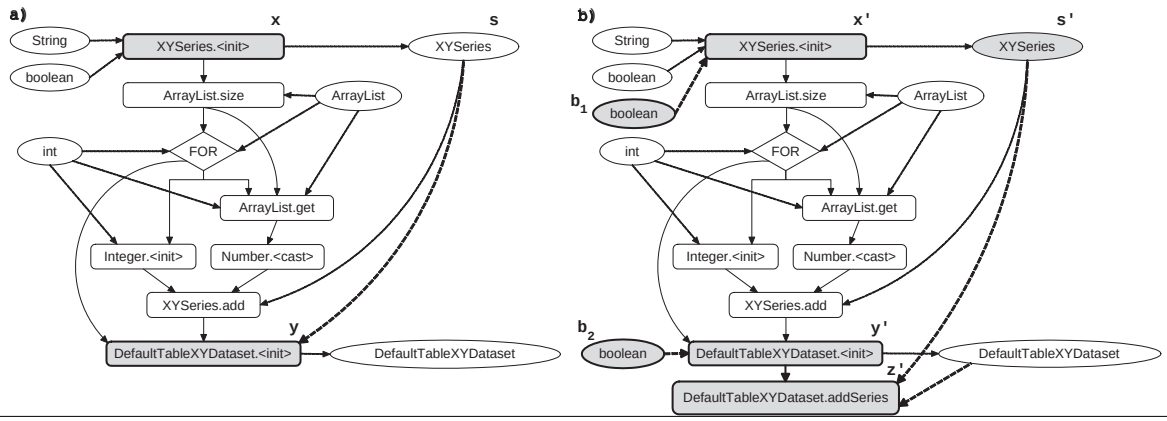

Figure 7. API i-Usage models in JBoss before and after migration to a new JFreeChart library version

Figure 7 shows two graph-based API usage models extracted from the code in Figure 1. The usage changes between two models are illustrated by the gray nodes with bold edges. For simplicity, in the figure, a label is displayed with only class and method names, even though our model actually retains the fully qualified class name and the signature of a method. In Figure 7b, an edge from the action node y'=DefaultTableXYDataset.<init> to the action node z'=DefaultTableXYDataset.addSeries represents that y' is used before z'. An edge from the action node x=XYSeries.<init> to the data node s=XYSeries shows that s is used to store the output of x. An edge coming out of s changes its target from y to z'. That means, s' is now used as an input to z' instead of y'. Note that x and x' represent different API elements—x is a deprecated constructor with two parameters while x' is a new constructor with three parameters. The figure also shows a for loop related to the invocation of method XYSeries.add.

4.1.2 i-Usage Extraction

CUE extends our prior work (Nguyen *et al.*'s graph-based object usage model [25]) to build API usage models from each method in client code. It parses the source code into Abstract Syntax Trees (AST), traverses the trees to analyze the AST nodes of interest within a method such as method invocations, object declarations and initializations, and control statements (e.g. if, while, for), and builds the corresponding action, data, and control nodes along with control and data dependencies between them. Static methods, type casting, and type checking operations of a class are considered as special invocations of the corresponding objects. After extraction, CUE removes all action and data nodes and the edges that do not represent the usages of API elements or have no dependencies with those API elements. In other words, CUE determines a sub-graph of the original object usage model that is relevant to the usage of API elements by performing program slicing from the API usage nodes via control and data dependency edges. Moreover, since a particular API could be used by multiple methods in client, CUE uses a set of iGROUM models to represent API i-usages.

While building an iGROUM, CUE also takes into account subtyping information, which is described in details in Section 4.2. CUE uses the inheritance information of the system to create nodes and labels more precisely. For example, if a method $C.m$ is called in an iGROUM, CUE checks whether $C.m$ is inherited from a method $A.m$, i.e., $C.m$ is *not explicitly* declared in the body of the class C. If that is the case, the action node corresponding to the call would be a node with the label built from $A.m$, rather than from $C.m$. If $C.m$ overrides $A.m$, the label is built from $C.m$.

Furthermore, CUE also performs an intra-procedural analysis on object instantiation, assignment, and type casting statements to keep track of the types of variables used within a method. For example, if it encounters a method call $o.m$ with o being an object declared with type C, and later finds that o is casted into an object of class C', then the label of action node for $o.m$ is built from $C'.m$, rather than $C.m$.

4.2 API Usage via Inheritance

This section presents our graph-based representation for API usages via inheritance and the corresponding extraction.

4.2.1 Method Overriding and Inheritance

Assume that class C in a client code directly inherits from an API class A. Method $C.m$ **overrides** a non-static method $A.m$ if $C.m$ is declared in class C and has the same signature with $A.m$. In Object-Oriented Programming, method $A.m$ is not considered to be overridden in C when the method $C.m$ with the same signature as $A.m$ is not *explicitly* declared in C. However, because $C.m$ could still be invoked, CUE still considers that $C.m$ exists and **inherits** from $A.m$. If $A.m$ and $C.m$ are static, CUE does not consider that $C.m$ overrides $A.m$ because they are called based on the declaring types. If $A.m$ is static and $C.m$ is not explicitly declared in C, CUE does not consider the existence of $C.m$.

1. If $C.m$ inherits $A.m$, a call to $C.m$ will be a call to $A.m$. Thus, if $A.m$ is changed, not only the calls to $A.m$ need to be adapted in response to the change of $A.m$, but also all the calls to $C.m$ need to be considered for adaptation.

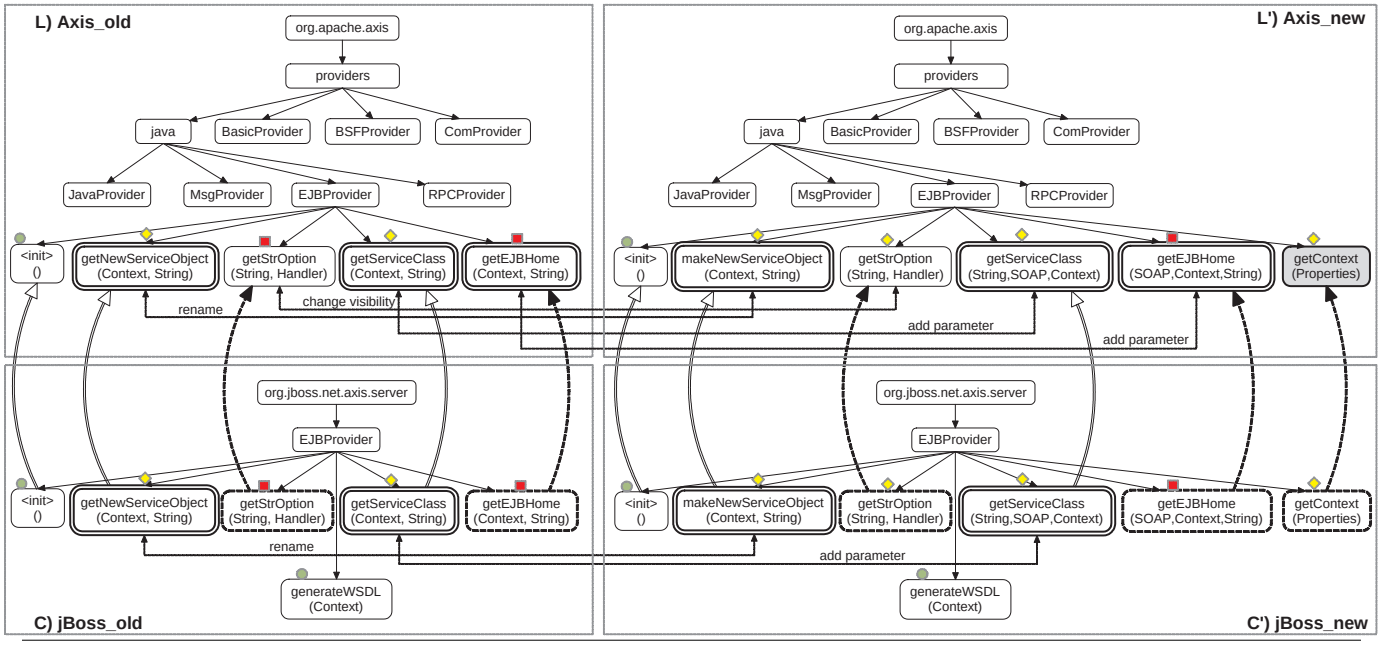

Figure 8. API x-Usage models in JBoss before and after migration to a new Axis library version

For example, if $A.m$ has a newly added parameter, all method calls to $A.m$ and $C.m$ must be considered for the adaptation of adding a new parameter. Otherwise, the program might not be compiled or such calls might be accidentally dispatched as a call to another method that has the same-signature as $A.m$ (e.g. a parent method $A_o.m$ of $A.m$).

2. If $C.m$ overrides $A.m$, they need to have the same signature. Thus, if $A.m$ is changed, $C.m$ needs to be considered to be changed correspondingly (see Figures 3 and 4).
3. A call to $A.m$ might be a call to $C.m$ in run-time due to dynamic dispatching. Thus, if $C.m$ is changed, not only all the calls to $C.m$ and $C_1.m$, with C_1 being a descendant class of C, are considered for adaptation correspondingly, but also all calls to $A.m$ must be taken into consideration.

In CUE, the overriding and inheritance relationships are defined in the same way as above among the methods of two API classes A and A_1 in which A_1 inherits from A, and among the methods of two client classes C and C_1 in which C_1 inherits from C.

4.2.2 x-Usage Model

Now, let us describe the model and extraction algorithm for API usages via inheritance. CUE uses xGROUM (Extension-based, GRaph-based Object Usage Model) to represent all API x-usages in the client system and and all libraries by considering each library a sub-system of the client system under investigation.

An xGROUM is a directed, labeled, acyclic graph in which each node represents a class or a method in the client system and its libraries. The label of a node is its fully qualified name and signature. Interfaces are considered as special classes.

Edges between class nodes represent subtyping relations. Edges from class nodes to method nodes represent the containment relations. Between method nodes, there are two kinds of edges: o-edge (**overriding**) and i-edge (**inheriting**):

- An o-edge from a node $C.m$ to $A.m$ shows that $C.m$ overrides $A.m$. This means that C inherits from A, and $C.m$ is declared in C and has the same signature as $A.m$.
- An i-edge from a node $C.m$ to $A.m$ shows that $C.m$ inherits from $A.m$. This means that C inherits from A, and $C.m$ is *not explicitly declared* in C even though $C.m$ could be invoked and has the same signature as $A.m$. $C.m$ is called an i-node in xGROUM, and other method nodes are called o-nodes.

Figure 8 illustrates the xGROUM for Figure 4. Figures 8L and 8C show the API class A=EJBProvider in package org.apache.axis.providers.java and the client class C=EJBProvider in package org.jboss.net.axis.server. Figures 8L' and 8C' show those two classes in their new versions. The o-edges and o-nodes, such as C.getNewServiceObject, are illustrated with solid double lines, meaning that they are of interest. The i-nodes such as C.getEJBHome and the i-edges are shown in dashed lines, meaning that they are just placeholders and not being really declared or created.

Added nodes such as A.getContext(Properties) are painted in gray color. Updated nodes are represented in double lines along with bi-directional arrows between them in the graphs of two versions. For example, an arrow with the la-

bel rename from node A.getNewServiceObject (in Figure 8L) to A.makeNewServiceObject (in Figure 8L') shows that those two nodes represent a renamed method. An arrow from node A.getServiceClass(Context, String) in Figure 8L to node A.getServiceClass(String, SOAP, Context) in Figure 8L' signifies the change in the parameter list of the corresponding method. As shown in Figure 8C', class C in JBoss is adapted accordingly to the changes to class A in Axis.

To build xGROUM, CUE extends the inheritance hierarchy by adding o-edges between methods. The i-nodes and i-edges are not explicitly created, but being computed on demand. Note that only one xGROUM is built for the entire client system and its libraries.

5. Usage Adaptation Miner (SAM)

This section describes our API usage adaptation miner, SAM. SAM uses iGROUMs to represent API i-usages in any client code as well as in the library's test and demo code. Thus, the adaptation of API usages could be modeled as a generalization of changes to the corresponding individual iGROUMs. SAM also uses a graph alignment algorithm to identify API i-usage changes that are caused by changes to APIs, and a mining algorithm that generalizes common edit operations from multiple API usage changes to find API usage adaptation patterns. LIBSYNC uses such patterns to recommend the locations and edit operations.

5.1 i-Usage Change Detection

Using OAT, LIBSYNC derives sets ΔL and ΔP containing the changed entities (including packages, classes, methods) of the library and the client program respectively. It is able to align such code entities between two versions as well. Thus, for any method $m \in \Delta P$, LIBSYNC builds two iGROUMs U and U' for m in two corresponding versions. Then, it uses GroumDiff, our graph-based alignment and differencing algorithm, to find the changes between the corresponding usage models U and U'.

Our graph alignment algorithm, GroumDiff, maps the nodes between two iGROUMs such that the aligned nodes represent the unchanged, updated, or replaced nodes while unmapped nodes represent the added/deleted nodes. Let us illustrate the pseudo-code of our GroumDiff algorithm (Figure 9) via the example in Figure 7. The mapped nodes in Figure 7 would be the ones with identical labels (e.g. x and x', y and y'). New nodes like z', b_1, b_2 would not be mapped. As we could see, mapped nodes tend to have highly similar labels and structures. For example, unchanged API elements would have identical names; replaced ones tend to have similar names; and both types tend to have similar neighborhood structure with the others.

The idea of GroumDiff algorithm is to map the nodes between two graphs based on the similarity of their labels and neighborhood structures with other nodes. The similarity of node labels, $lsim(u, v)$, is based on string-based

```
1  function GroumDiff(U, U') // align and differ two usage models
2   for all u ∈ U, v ∈ U' // calculate similarity between u and v
        based on label and structure
3    sim(u, v) = α • lsim(u, v) + β • nsim(u, v)
4   M =MaximumWeightedMatching(U, U', sim) // matching
5   for each (u, v) ∈ M:
6    if sim(u, v) < λ then M.remove((u, v)) //remove low matches
7    else switch // derive change operations on nodes
8     case Attr(u) ≠ Attr(v): Op(u) = Op(v) = ''replaced''
9     case Attr(u) = Attr(v), nsim(u, v) < 1: Op(u)=''updated''
10    default: Op(u) = ''unchanged''
11  for each u ∈ U, u ∉ M: Op(u) = ''deleted'' // unaligned nodes
12  for each v ∈ U', v ∉ M: Op(v) = ''added'' // are deleted/added
13  Ed = EditScript(Op)
14 return M, Op, Ed
```

Figure 9. API Usage Graph Alignment Algorithm

Levenshtein distance [17]. It also takes into account the renamed API elements: the similarity level of the labels representing renamed or moved API elements is set as high as that for unchanged ones. Neighborhood structures of nodes is approximated by Exas characteristic vectors [24], thus, their similarity measurement $nsim(u, v)$ is based on the distance of such vectors. GroumDiff calculates and combines the similarity of labels and neighborhood structures of all pairs of nodes u and v between two graphs as $sim(u, v) = \alpha \bullet lsim(u, v) + \beta \bullet nsim(u, v)$ (line 3). Since each node u in a graph should be mapped to only one node v that has the highest possible similarity, GroumDiff finds the maximum weighted matching on such nodes using the calculated similarity values as weights (line 4). The resulting pairs of matched nodes are the alignment results (lines 7-10). Matched nodes having little similarity would be reported as unmapped nodes (lines 11-12).

Then based on the alignment results, SAM derives a sequence of graph edit operations. That is, the un-aligned (unmapped) nodes are considered *added or deleted* (lines 11-12). Aligned nodes with different labels, or the same labels but different structures or attributes are considered *updated or replaced* (lines 8-9). Other nodes are considered *unchanged* (line 10). From this information, GroumDiff derives an edit script to describe the changes as a sequence of graph operations (line 13). This edit script is then used to mine the API usage adaptation patterns.

Let us revisit Figure 7. GroumDiff aligns nodes with identical names in Figures 7a and 7b. Node z' = DefaultTableXYDataset.addSeries and two nodes, boolean b_1 and b_2, are not mapped; thus, they are considered as added. The nodes with the label <init> (x and x') are replaced. The node s=XYSeries is updated because its neighboring nodes changed. Thus, the derived editing script is

```
Replace XYSeries.<init>(..., boolean)
        XYSeries.<init>(..., boolean, boolean)
```

```
protected JFreeChart createXyLineChart() throws JRException {
  ChartFactory.setChartTheme(StandardChartTheme.createLegacyTheme());
JFreeChart jfreeChart=ChartFactory.createXYLineChart(..., getDataset(),...);
 ...
return jfreeChart
}
```

Figure 10. API Usage Changes in JasperReports

```
Replace DefaultTableXYDataset.<init>(XYSeries)
        DefaultTableXYDataset.<init>(boolean)

Add DefaultTableXYDataset.addSeries(XYSeries)
```

Improvement. To improve the alignment accuracy and to deal with renamed nodes, SAM uses OAT to find API methods and classes whose declaration changed. Then, it makes the action nodes representing the calls to them to have the same labels in two iGROUMs under comparison. That is, if m is updated into m' in the library through renaming, the label of an action node representing an invocation of m' is replaced by the label built from m. Note that those two nodes must also have similar neighborhoods. In brief, SAM uses the knowledge of the origin analysis result to improve the alignment of nodes in the corresponding iGROUMs.

5.2 x-Usage Change Detection

Changes to an xGROUM are detected by OAT and represented as editing operations: (1) Add/Delete nodes and edges: e.g., a new class is added, a method is deleted, or an overriding edge changes its target method; (2) Replace/Update nodes and edges: e.g an edge is changed from i-edge to o-edge when a new method overrides a parent method. It is changed from o-edge to i-edge when an overriding method is deleted.

Note that when the signature of a method $C.m$ is changed into $C.m'$ that overrides some parent method $A.m'$, SAM considers this change as the addition of a new o-node for $C.m'$, the old node $C.m$ having the same signature with $A.m$ will become an i-node.

5.3 Usage Adaptation Pattern Mining

Given a library L and a client system P, SAM identifies the locations and edit operations required to adapt API usages when migrating to the version i of L. Since individual API usages can have different edit operations between two corresponding iGROUMs, SAM finds a common subset of edit operations that occur frequently among multiple API usages. We call such frequent edit operations an *adaptation pattern*.

For example, JasperReports version 3.5.0 migrated to use JFreeChart API version 1.0.12. Analyzing JasperReports' code, we found that the addition of the invocation statement ChartFactory.setChartTheme(StandardChartTheme.createLegacyTheme()); before the call to ChartFactory.create*Chart occurs in 53 methods (Figure 10). That is, JFreeChart at version 1.0.12 has a new feature, which specifies the style or theme of a chart object. This new feature requires that the instantiation of a chart object needs to create a ChartTheme object first. JFreeChart's ChartFactory, the factory class for creating chart objects, now has a new method ChartFactory.setChartTheme to set the theme for a chart object. JFreeChart also provides a class StandardChartTheme as the default implementation of ChartTheme which has a method named StandardChartTheme.createLegacyTheme() to create and return a ChartTheme that does not apply any changes to the JFreeChart defaults.

```
1 function ChangePattern(ΔP_i, ΔL_i) //mine usage change patterns
2   for each (U, U') ∈ UsageChange(ΔP_i, ΔL_i) //compute changes
3     Add(GroumDiff(U, U')) into E // add to dataset of sets of ops
4   F = MaximalFrequentSet(E, σ) //mine maximal frequent subset
        of edit operations
5   for each f ∈ F:
6     Find U, U' : f ⊂ GroumDiff(U, U') //find usages changed by f
7     Extract (U_o(f), U'_o(f)) from (U, U') // extract ref models
8     Add (U_o(f), U'_o(f)) into Ref(f) // add to reference set for f
9 return F, Ref
```

Figure 11. Adaptation Pattern Mining Algorithm

Mining Algorithm. The algorithm to recover the API i-usage adaptation patterns is illustrated in Figure 11. It receives two inputs: (1) ΔL_i is the set of API elements that changed at or before version i and (2) ΔP_i contains program entities in client code that changed after migration to the version i of L. The change set, ΔL_i, is computed by applying OAT to the version history of the library L backward from the version i. Similarly, ΔP_i is computed by running OAT on two versions of the client program before and after migration to L_i.

The first step is to determine all API i-usages that changed due to the changes ΔL_i (UsageChange(ΔP_i,ΔL_i) in line 2). This step is necessary since some i-usage changes are irrelevant to the API changes. To do that, SAM determines all methods in both L and P that are affected by the change in L_i by using the information produced from the location detection algorithm (Section 6.2.1 will detail this algorithm). More specifically, it uses the output of that algorithm, i.e. the change set $IU(P, \Delta L_i)$ that contains the methods and classes in the client code and library that are affected by the API's changes at version i via method overriding and inheritance relations. Then, SAM removes API i-usage changes that have nothing to do with the set $IU(P, \Delta L_i)$.

Next, for each of such usage changes, SAM extracts from the corresponding surrounding code the pairs of usage models (U, U') before and after the change at version i (line 2). To do this, for each changed method $m \in \Delta P_i$ containing such usage changes, SAM builds the corresponding usage models (U, U'), and determines whether U and/or U' have any action nodes that represent any method(s) in the change set $IU(P, \Delta L_i)$. If such a pair exists, their changes would be

related to the API changes. SAM uses GroumDiff to find the changes between U and U' in term of a set of graph editing operations. That set of operations is added into the set E of API usage changes caused by the API's changes (line 3).

Then, SAM mines the maximal frequent subset of editing operations for all the sets in E, using the frequent itemset mining algorithm in [2]. This algorithm finds every set f that occurs in the sets in E with a relative frequency (i.e. confidence) at least σ, with σ is a chosen threshold, and with its size as large as possible (line 4). For each of such f, SAM finds all pairs (U, U') whose change operations include f (line 6). For each pair (U, U'), it extracts the usage skeletons $U_o(f)$ and $U'_o(f)$ (line 7). This pair of usage skeletons are called *reference models*, which provide the context of the change f (will be explained next). Then, it adds that pair into a set $Ref(f)$ for each mined frequent subset f (line 8), which is now considered as an adaptation pattern of API usages (i.e. frequent changes on API usage models).

Relative frequency. The relative frequency of a set of change operation f is calculated as follows. Assume that f is a subset of edit operations from U to U'. $Freq(f)$ denotes the frequency of f, i.e. the number of sets of change operations in E contain f. $NUsage(f)$ is the number of API usages of the nodes affected by f, i.e. the number of all iGROUMs containing $U(f)$. Then, the relative frequency of f is defined as $Freq(f)/NUsage(f)$.

Reference model. $U_o(f)$ is defined as the set of mapped nodes in U that are affected by f and their dependent nodes via control and data dependencies. $U'_o(f)$ is similarly defined. $U_o(f)$ and $U'_o(f)$ provide the contextual information on the change f. Thus, they are called the reference models of f. If another usage V contains $U_o(f)$, one could consider that V also has a context that could be adapted by the frequent adaptation f. Thus, $U_o(f)$ and $U'_o(f)$ are used to model the usage skeletons corresponding to the adaptation pattern f.

Figure 12 shows an adaptation pattern and its reference models found in JasperReports with respect to JFreeChart library migration. The pattern includes the addition of two method calls StandardChartTheme.createLegacyTheme and ChartFactory.SetChartTheme, which lead to the addition of two new action nodes and one data node, along with the associated edges, and the addition of an edge from the data node ChartFactory. Since setChartTheme and createAreaChart use the same data node ChartFactory, SAM derives the reference models of this change as U_0 and U'_0 as in Figure 12. U'_0 contains not only the added sub-graph but also the nodes having dependencies with the changed nodes.

The use of reference model is also useful in the cases of *newly added* API elements. Suppose that m is a newly added method in the new version of a library and a call to m is added in U'. In this case, no node in U can be mapped to the data node m. However, there might have some other currently existing nodes that are dependent on m and they could be mapped back to U. Thus, SAM could use those nodes as referenced nodes for mapping between U and U' in the case of newly added nodes. In such cases, SAM will also add those dependent nodes into the reference model for later mapping.

Improvements. To improve the accuracy of the mined patterns of usage changes, ΔP_i contains the code taken from different sources: client code on different systems, or test code and demo code provided inside the API's source code. The threshold σ will be slightly different. For example, on test code and demo code, a usage pattern might be tested or demonstrated for only once. Therefore, we could choose small σ. Test code might contain the initialization of test data and the assertion of test results, which might not be parts of API usage specifications. To improve the quality of mined protocols, SAM discards such initializations and assertions when building the iGROUMs on the test code.

6. Recommending Adaptations

Sections 6.1 and 6.2 discuss how LIBSYNC suggests the code locations to be adapted and edit operations required for those API usage adaptations.

6.1 API i-Usage Adaptation Recommendation

After detecting API changes via OAT and mining usage adaptation patterns on relevant codebases via SAM, LIBSYNC has a knowledge base of API usage skeletons and corresponding adaptation patterns for an API L of interest. For each version i of the library L, the knowledge base contains the set of usage adaptation patterns F at that version. Each pattern f has a set of reference usage models $Ref(f) = (U_0, U'_0)$. It also contains ΔL_i, the set of changed entities of L from any two consecutive versions. With this knowledge, LIBSYNC provides API usage adaptation recommendations on any given client code Q that needs to be adapted to a version i of L.

6.1.1 Location Recommendation

First, LIBSYNC determines the code locations in the client system Q that potentially need adaptation to L_i. Using ΔL_i and xGROUM model of Q at that version, LIBSYNC computes two change sets of methods $XU(Q, \Delta L_i)$ and $IU(Q, \Delta L_i)$ that are potentially affected by the changed entities in ΔL_i. Details of the method to derive those two change sets will be explained in Section 6.2.2. $IU(Q, \Delta L_i)$ is the set of methods and classes in L and Q that are affected by changed entities in ΔL_i (including overridden and inherited methods). $XU(Q, \Delta L_i)$ is the set of methods and classes in Q that are affected by the changed entities in ΔL_i via method overriding and inheritance. Every code location that uses an entity in $IU(Q, \Delta L_i)$ will be considered for adaptation to the changes ΔL_i of L. We use $AU(Q, \Delta L_i)$ to denote the set of the corresponding iGROUMs of such code locations.

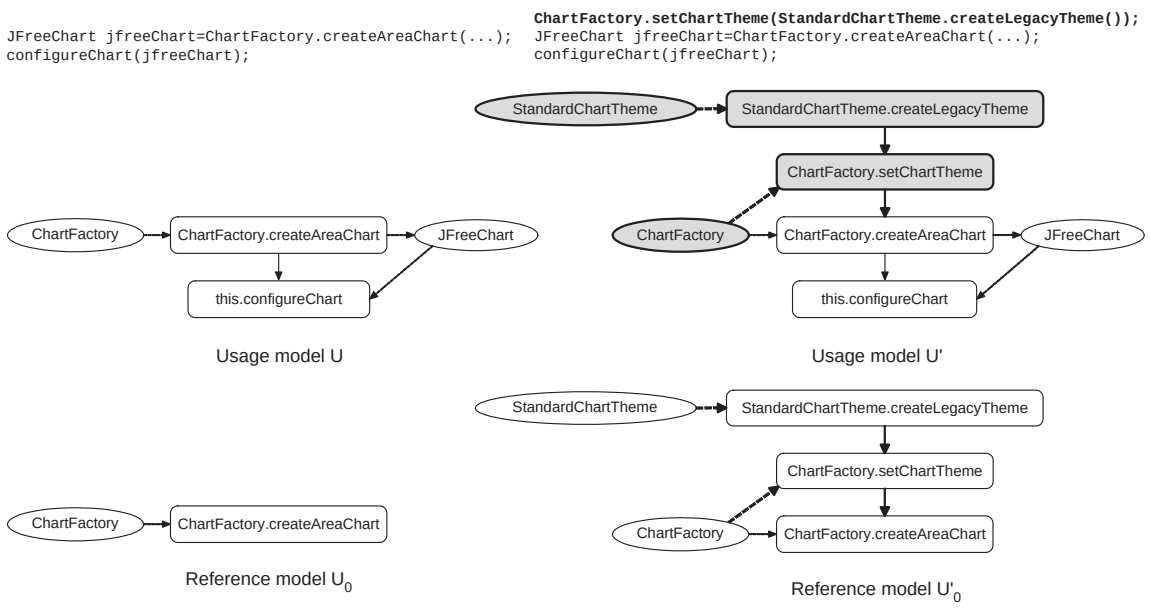

Figure 12. API Usage Change Patterns and Reference Models

To improve the performance, LIBSYNC uses some preprocessing techniques. Based on text-based filtering, it finds the source files that could involve the usages of L. Each source file is tokenized. If a file does not contain any token similar to the names of classes/methods in $IU(Q, \Delta L_i)$, it will be discarded from further processing. In the next step, the remaining source files are parsed and extracted to build API i-usage models. For each model V, LIBSYNC checks whether V contains some nodes representing the usages of any entity in $IU(Q, \Delta L_i)$. If that is the case, it will report V as a location for consideration of adaptation, i.e. V will be added to $AU(Q, \Delta L_i)$.

Let us revisit the example in Figures 1 and 7: L = JFreeChart, i = 0.9.17, Q = JBoss 3.2.7. Assume that JBoss is currently using JFreeChart 0.9.15. Using OAT, LIBSYNC could detect $IU(Q, \Delta L_i) = \{A, B, x, x', y, y'\}$ with the following information:

Id	Label	Change
A	XYSeries	modified class
B	DefaultTableXYDataset	modified class
x	XYSeries.<init>(String,boolean)	deprecated
x'	XYSeries.<init>(String, boolean, boolean)	added
y	DefaultTableXYDataset.<init>(XYSeries)	deprecated
y'	DefaultTableXYDataset.<init>(boolean)	added

Using text-based filtering, LIBSYNC detects that the source file ManageSnapshotServlet.java in Q = JBoss 3.2.7 has some tokens XYSeries and DefaultTableXYDataset. Extracting iGROUMs from this file for further analyzing, it finds that the iGROUM V of method doit has the nodes whose labels appear in $IU(Q, \Delta L_i)$ (Figure 7). Thus, it reports V as a code location that may need the adaptation, and adds V to $AU(Q, \Delta L_i)$ for further operation recommendation.

```
1  function Adapt(V, F) //adapt API usage based on change patterns
2    for each U_o ∈ Ref(F): //for each change pattern f: calculate
         similarity to reference models
3    Relevant(V, U_o) = sim(GroumDiff(V, U_o))
4    (f*, U_o*) = Max(Relevant) //find the most suitable
5    Ed = GroumDiff(U_o*, U_o'*) //derive referenced change operations
6    Recommend(Ed, V) //and recommend
```

Figure 13. Usage Adaptation Recommending Algorithm

6.1.2 Operation Recommendation

LIBSYNC uses the API i-usage change patterns in its knowledge base to derive the recommended operations for each iGROUM V in the set $AU(Q, \Delta L_i)$ of usage models that are considered for adaptation. Figure 13 illustrates the algorithm for this task. First, LIBSYNC determines the change pattern $f*$ that is most suitable for V (lines 2-3). For each pair of reference models (U_o, U'_o) in the set of all reference models in the knowledge base $Ref(F)$, LIBSYNC maps U_o and V using the GroumDiff algorithm (Figure 9), and computes the relevant degree between V and U_o based on the number of matched nodes over the size of U_o (line 4). Next, it ranks them to find the reference model U_o* that is best matched to V (i.e. with highest relevance) (line 4). At last, LIBSYNC finds the changes of the best matched reference model $U*$ (line 5) and recommends such changes as edit operations on iGROUM V (line 6).

Notes. Since a usage model could use many usage protocols, LIBSYNC may find more than one usage change patterns f that could be mapped against V. Thus, it ranks them based on their similarity with V and their frequencies (the higher the frequency is, the more correct the recommenda-

tion would be). If no change pattern is suitable (e.g. the similarity is too little), V will be considered as an API usage irrelevant to API changes in ΔL_i.

After processing all usage models, for each model V in recommended list $AU(Q, \Delta L_i)$, LIBSYNC reports its location, its ranked usage adaptation patterns fs (with similarity levels and frequencies). It also provides with each pattern a code skeleton that was collected during the usage pattern mining process. If users choose a code location and a usage change pattern for adaptation, LIBSYNC provides the recommendation for adaptation at that location.

Let us revisit the example in Figures 10 and 12 for L = JFreeChart, i = 1.0.12, Q = JasperReports 3.5.0. LIBSYNC detects the changed set ΔL_i of JFreeChart at that version as:

Id	Label	Change
A	StandardChartTheme	added class
B	ChartFactory	added class
a	StandardChartTheme.createLegacyTheme	added
b	ChartFactory.setChartTheme	added

Mining on the code base P = JasperReports, LIBSYNC recovers the change pattern f = [Add a, Add b] with 53 pairs of reference models (one pair is the iGROUMs (U, U') for code fragments in Figure 10). In Q, LIBSYNC determines that iGROUM V uses a method of class ChartFactory. Since ChartFactory is in ΔL_i, it is put into $IU(Q, \Delta L_i)$, and thus, V is put into $AU(Q, \Delta L_i)$, meaning that it should be adapted.

Matching V with the change patterns and reference models, LIBSYNC finds U as the best match for V with the change pattern f. In the matching, it also finds the maps between the action nodes for two method calls c and d with the label ChartFactory.createXYLineChart in U and V. Differencing U and U' gives the operations Ed = [Add a, Add b]. Thus, LIBSYNC recommends to add those two method calls, a and b, into V, along with their dependencies: a is called before b and the output of a is the input of b; b is called before c due to such dependencies in U. To help in adaptation, LIBSYNC also provides the reference code in JasperReports (Figure 10).

6.2 API x-Usage Adaptation Recommendation

This section describes how LIBSYNC recommends adaptations for inheritance-based library usages.

6.2.1 Location Recommendation

To find the changes of xGROUM and recommend relevant adaptation, LIBSYNC starts with the change set of API ΔL and the change set ΔP of classes and methods in the client code. Those two change sets are obtained from the execution of OAT on two versions of both API and client sides.

The outputs of this location recommendation algorithm are two change sets $XU(P, \Delta L)$ and $IU(P, \Delta L)$ of classes and methods that would be affected by the changes in ΔL in the API, taking into account x-usages and i-usages respectively. Thus, they are also classes and methods that could need the adaptation. This algorithm is carried out as follows:

- If $A.m \in \Delta L$, any method $C.m$ overriding $A.m$ is considered to be adapted. Thus, as $A.m$ changes, $C.m$ is added into $XU(P, \Delta L)$. $C.m$ is also added into $IU(P, \Delta L)$ for the consideration of usage adaptation.
- If $A.m \in \Delta L$, any method $D.m$ inheriting $A.m$ is also considered for adaptation for API usages via invocation, i.e. $D.m$ is added into $IU(P, \Delta L)$, because a method call to $D.m$ could be actually a call to $A.m$.
- If $A.m \in \Delta L$, and if $C.m \in \Delta P$ and $C.m$ overrides $A.m$, then $A.m$ and any ancestor method $A_o.m$ of $A.m$ (i.e. overridden or inherited) is also considered to be adapted (i.e. $A.m$ and $A_o.m$ are added to $IU(P, \Delta L)$), because a call to $A_o.m$ or $A.m$ might be dynamically dispatched as a call to $C.m$.

Let us take an example with P = JBoss, L = Axis. The changes are in Figures 3, 4, and 8. The set ΔL contains the following classes and methods:

Id	Label	Change
A	EJBProvider	modified class in Axis
B	Serializer	modified class in Axis
$A.n$	EJBProvider.getNewServiceObject	renamed
$A.p$	EJBProvider.getContext	added
$A.q$	EJBProvider.getEJBHome	changed in parameter
$B.m$	Serializer.writeSchema	changed in parameters, RetType
...	...	

Then, based on the xGROUM, two methods $D.m$ and $E.m$ (overriding $B.m$) and the method $C.n$ (overriding $A.n$) are considered to be adapted, i.e. added to $XU(P, \Delta L)$ (see the Table below for the ids). Their corresponding classes are also added to $XU(P, \Delta L)$. Thus, the set $XU(P, \Delta L)$ contains the following classes/methods:

Id	Label	Change
C	EJBProvider	extend modified class in JBoss
D	AttributeSerializer	extend modified class in JBoss
E	ObjectNameSerializer	extend modified class in JBoss
$C.n$	EJBProvider.getNewServiceObject	should be renamed
$C.q$	EJBProvider.getEJBHome	should be changed in paras
$D.m$	AttributeSerializer.writeSchema	change in paras, RetType
$E.m$	ObjectNameSerializer.writeSchema	change in paras, RetType
...	...	

They are also added to $IU(P, \Delta L)$, along with $A.p$ (newly added method) and other i-nodes, i.e. the placeholders such as $C.p$. The $IU(P, \Delta L)$ set contains the followings:

Id	Label	Change
A	EJBProvider	modified class in Axis
B	Serializer	modified class
C	EJBProvider	modified class in JBoss
D	AttributeSerializer	modified class in JBoss
E	ObjectNameSerializer	modified class in JBoss
$A.n$	EJBProvider.getNewServiceObject	renamed
$C.n$	EJBProvider.getNewServiceObject	should be renamed
$A.p$	EJBProvider.getContext	added
$C.p$	EJBProvider.getEJBHome	inherited from added method
$C.q$	EJBProvider.getEJBHome	should be changed in paras
$D.m$	AttributeSerializer.writeSchema	change in paras, RetType
$E.m$	ObjectNameSerializer.writeSchema	change in paras, RetType
...	...	

The outputs $XU(P, \Delta L)$ and $IU(P, \Delta L)$ are used in the mining algorithm (Figure 11), in location/operation recommendation for API i-usages (Section 6.1.1), and operation recommendation for x-usages (Section 6.2.2).

6.2.2 Operation Recommendation

After detecting $XU(P, \Delta L)$, LIBSYNC will recommend for adaptation of API x-usages for the methods in $XU(P, \Delta L)$. Currently, the recommendation for x-usages is as follows:

- Pointing out the classes/methods that need API x-usage adaptation. For example, two methods AttributeSerializer.writeSchema and ObjectNameSerializer.writeSchema in Figure 3.
- Showing the changes to the API classes and methods in use. For example, it shows the changes to Serializer.writeSchema with two operations: Add a new parameter and Replace the return type.
- Suggesting the operation of classes and methods in client code that need adaptation. For example, it suggests to Add a parameter of type Class, and to Replace return type into org.w3c.dom.Element. It recommends fully qualified names to help the developers to use correct packages.

7. Evaluation

This section presents the evaluation of our framework. For OAT, the parameter setting, $\delta = 0.75$ and $\mu = 0.625$, is used. For SAM, the parameter setting, $\alpha = 0.5$, $\beta = 0.5$ and $\lambda = 0.5$ is used in GroumDiff algorithm, and $\sigma = 0.5$ is used in ChangePattern.

7.1 Precision and Recall of Origin Analysis

To evaluate the quality of change detection in OAT, we conducted two experiments. First, we manually checked the results. Second, we compared our results with Kim *et al.*'s API matching results [18], which has been compared with a number of origin analysis tools [19, 38, 40, 41]. In the first experiment, we executed OAT on four different version pairs of JHotDraw (see Table 1). JHotDraw was chosen due to the availability of source code and its rich set of documentation.

The result is shown in Table 1. Columns Mapped and Checked show the numbers of method-level matches that were returned from our tool OAT and the ones that were manually checked respectively. Between two versions 5.4b2-6.0b1, OAT returned 3,250 pairs, and we checked randomly-selected 100 pairs. Columns $\surd$ and X display the correctly and incorrectly detected matches. Precision shows the precision value, which is the ratio between the number of correctly detected pairs over the total number of checked, detected pairs. The precision of OAT is very high with only a couple of incorrect pairs. For two versions 5.4b1-5.4b2, there are only 9 method-level matches because those versions are beta releases with minor changes. In other cases, OAT's precision ranges from 97% to 100%.

In the second experiment, we executed both OAT and Kim's tool on several consecutive version pairs of JFreeChart and JHotDraw. From the outputs of two tools, all method-level matches were compared to find the common set of matches (column $\bigcap$), and to identify a set of matches that were returned by OAT but not by Kim's (column OAT-Kim), and a set of matches that were found by Kim's but not by OAT (column Kim-OAT). Those differences were further manually checked to see if they are correct (column $\surd$), incorrect (column X) or undecidable (column ?). For each group, we also computed the number of correct pairs and incorrect pairs over the total number of pairs: column TP (True Positive) and FP (False Positive) respectively. Table 2 shows the comparison results. On average, OAT reports fewer pairs but its accuracy is often higher. In addition, the gap between $(TP + FP)$ to 100% is smaller in OAT than Kim's tool. This means that OAT produced fewer cases that were hard to determine the correctness of the involved matches.

Table 1. Precision of Origin Analysis Tool OAT

Version Pairs	Mapped	Checked	$\surd$	X	Precision
5.2-5.3	71	71	69	2	97%
5.3-5.4b1	70	70	68	2	97%
5.4b1-5.4b2	9	9	8	1	89%
5.4b2-6.0b1	3,250	100	100	0	100%

7.2 Adaptation of i-Usage

We evaluated the quality of LIBSYNC in recommending API i-usage adaptations. In order to recommend API i-usage adaptation, LIBSYNC needs to detect API i-usage changes and derive adaptation patterns.

The experiments were carried out on large-scale, real-world systems in different application domains with long histories of development. Table 3 shows the details about those subject systems. For example, JBoss is a middle-ware framework that has been developed for more than 6 years with more than 40 releases. It has about 40 thousand methods and uses hundreds of different libraries.

7.2.1 Detection of i-Usage Changes

In this experiment, our evaluation questions are (1) can CUE detect API usage changes correctly? and (2) are the client-side, API usage changes detected by CUE and SAM indeed caused by the evolution of libraries used by the client?

We ran our tool on those three subject client systems to report all API usage changes along with edit operations. For each client, we randomly picked 30 to 40 of the API usage changes. We manually checked the correctness of detected edit operations in API usage skeletons. In addition, we also examined whether the identified API usage changes are indeed caused by the changes to APIs.

Table 4 shows the result of this investigation. Column Changes shows the number of checked cases in detected API usage changes. Column Libs shows the number of libraries involved in those reported usage changes. The next two columns (Operations) display the numbers of correctly (see column $\surd$) and incorrectly detected API i-usage changes (column X) respectively. Similarly, the last two columns (column API) show how correctly our tool relates an API usage change to the changes to API(s).

Table 2. Comparison of Origin Analysis Tools

JFreeChart															
Pairs	OAT	Kim	∩	OAT - Kim						Kim - OAT					
				∑	√	X	?	TP	FP	∑	√	X	?	TP	FP
0.9.5-0.9.6	5	5	5	0	0	0	0	100%	0%	0	0	0	0	100%	0%
0.9.6-0.9.7	368	366	364	4	2	1	1	50%	25%	2	0	0	2	0%	0%
0.9.7-0.9.8	3157	3158	3121	36	36	0	0	100%	0%	37	7	30	0	19%	81%
0.9.9-0.9.10	144	159	130	14	3	10	1	21%	71%	29	14	2	13	48%	7%
0.9.10-0.9.11	9	7	7	2	2	0	0	100%	0%	0	0	0	0	100%	0%
0.9.11-0.9.12	66	66	35	31	12	10	9	39%	32%	31	19	6	6	61%	19%
0.9.12-0.9.13	134	133	133	1	1	0	0	100%	0%	0	0	0	0	100%	0%
0.9.13-0.9.14	84	96	74	10	6	3	1	60%	30%	22	12	6	4	55%	27%
0.9.14-0.9.15	6	12	6	0	0	0	0	100%	0%	6	6	0	0	100%	0%
0.9.15-0.9.16	79	75	65	14	13	0	1	93%	0%	10	2	4	4	20%	40%
0.9.16-0.9.17	205	240	171	34	4	30	0	12%	88%	69	27	42	0	39%	61%
0.9.17-0.9.18	36	45	36	0	0	0	0	100%	0%	9	0	9	0	0%	100%
0.9.18-0.9.19	140	282	102	38	30	8	0	79%	21%	180	41	139	0	23%	77%
Avg.	341.00	357.23	326.85	14.15	8.38	4.77	1.00	73%	21%	30.38	9.85	18.31	2.23	51%	32%
JHotDraw															
Pairs	OAT	Kim	∩	OAT - Kim						Kim - OAT					
				∑	√	X	?	TP	FP	∑	√	X	?	TP	FP
5.2-5.3	71	77	66	5	3	2	0	60%	40%	11	2	4	5	18%	36%
5.3-5.4b1	70	69	56	14	12	1	1	86%	7%	13	5	6	2	38%	46%
5.4b1-5.4b2	9	13	8	1	1	0	0	100%	0%	5	3	1	1	60%	20%
5.4b2-6.0b1	3,250	3,239	3,239	11	11	0	0	100%	0%	0	0	0	0	100%	0%
Avg.	850	849.5	842.25	7.75	6.75	0.75	0.25	86%	12%	7.25	2.5	2.75	2	54%	26%

Table 3. Subject Systems

Client	Life Cycle	Releases	Methods	Used APIs
JBoss (JB)	10/2003 - 05/2009	47	10-40K	45-262
JasperReports (JR)	01/2004 - 02/2010	56	1-11K	7-47
Spring (SP)	12/2005 - 06/2008	29	10-18K	45-262

Table 4. Precision of API Usage Change Detection

Client	Changes	Libs	Operations		API	
			√	X	√	X
JasperReports	30	5	30	0	27	3
JBoss	40	17	38	2	38	2
Spring	30	15	30	0	30	0

In most cases, our tool correctly detected the edit operations and correctly related the API usage changes on the client-side and the library-side changes (see two columns √). In 93 cases out of 100 checked cases, our tool correctly detected API usage changes and related them to library-side API declaration changes.

Example 1. Let us discuss an interesting case in Figure 14. This usage of JFreeChart creates a NumberAxis object and sets up its range and ticking unit. In the versions before 0.9.12 of JFreeChart, setting up the range of a NumberAxis object is carried out by invoking two methods setMinimumAxisValue and setMaximumAxisValue. However, from version 0.9.12, those two methods are deprecated, a new method setRange is added and should be used instead. SAM correctly identified API usage skeletons but did make some mistakes in deriving edit operations for adaptation. Instead of reporting two deletions and one addition, it reported one replacement and one addition. Importantly, however, SAM is able to recognize and correlate that the API usage change is due to the change in JFreeChart API specification.

```
NumberAxis yAxis = new NumberAxis(yTitle);
~~yAxis.setMinimumAxisValue(-0.2);~~
~~yAxis.setMaximumAxisValue(0.4);~~
yAxis.setRange(-0.2, 0.4);
DecimalFormat formatter = new DecimalFormat("0.##%");
yAxis.setTickUnit(new NumberTickUnit(0.05, formatter));
```

Figure 14. Create NumberAxis in jFreeChart

In some other cases, our tool wrongly related client-side updates with library-side updates even though the library-side updates did not affect the corresponding usage in the client code such as a method's access visibility modification. Another case is when the API method changes the types of exceptions that could be thrown, but the client code always catches the general exception type, Exception. Another one is when the API method changes the type of one parameter into its super-type (e.g. from String to Comparable). In those cases, there were some changes to those API usages but these changes were irrelevant to changes in the declaration of the API. Our tool mistakenly related them. Let us explain another interesting case of API usage changes due to the evolution of a library.

Example 2. Ruby, a scripting language/framework for Web applications, provides a new method parse in the version 0.8.0. This method accepts two string inputs: one referring to the piece of code required to compile and one referring to the compiling configuration. It returns a Node as the root node of the parse-tree. Using this newly added feature of Ruby, developers of Spring changed their implementation of the method createJRubyObject, which receives a string scriptSource as the input script, and returns an Object created by that script. In the old version of this method, it calls the evalScript method

```
...
 IRubyObject rubyObject = ruby.evalScript(scriptSource); //direct evaluatation
 if (rubyObject instanceof RubyNil) {
    throw new ScriptCompilationException(...);
...
```

```
...
 Node scriptRootNode = ruby.parse(scriptSource, ""); //parse the script
 IRubyObject rubyObject = ruby.eval(scriptRootNode); // eval the parse−tree
 if (rubyObject instanceof RubyNil) { //if cannot eval the whole script
   String className = findClassName(scriptRootNode); //just find class name
   rubyObject = ruby.evalScript("\n"+className+".new"); //to create an object
 }
 if (rubyObject instanceof RubyNil) {
    throw new ScriptCompilationException(...);
...
```

Figure 15. API usage changes in Spring with respect to the evolution of Ruby

directly on scriptSource. This direct evaluation could have a disadvantage in which the script is not well-formed, or more severely, is crafted as malicious code that exploits some vulnerabilities of the system. In the new version (Figure 15), Spring code first calls parse to parse the scriptSource into a tree, and then calls the eval method to execute this parsed code. If the script is ill-formed or maliciously crafted, the parsing will not return a well-formed parse tree and the eval method simply does not execute, thus, resolving the above vulnerability issue. LIBSYNC was able to mine this API usage adaptation pattern based on the API usage changes in the client code of Spring at 2.0 (JRubyScriptUtils.java).

7.2.2 Recommendation of Adaptation Locations

This section describes the evaluation of LIBSYNC in recommending the *code locations* for adaptation to a target library version. We chose six pairs of a library and its client. For each pair, let V_C and V_A be the versions of the client system and the library respectively. For each V_A, we selected another version V'_A of the library such that the client system had been changed in a later version than V_C. We ran LIBSYNC on V_A and V'_A to detect library-side changes and client-side API usage updates. LIBSYNC was run to recommend the locations for adaptation. Then, we manually checked in the history of the client code after that version V_C to see whether the code at those locations have been actually updated to work with the new library version V'_A.

Table 5 shows the result. JFree and Jasper are used as abbreviations for JFreeChart and JasperReports, respectively. Column Version shows the pairs of versions of the library and the client system. Column Rec. shows the number of the recommended locations. Columns $\surd$, X, Miss show the correctly, incorrectly, and missed detected locations respectively. Column Hint represents the cases in JasperReports corresponding to the changes of JFreeChart in which the API methods are *deprecated*, but developers have not updated yet. As shown, LIBSYNC provides highly correct locations. It missed in only one case out of 67 recommendation locations in total.

Table 5. Accuracy of i-Usage Location Recommendation

API - Client	Version	Rec.	$\surd$	Hint	X	Miss
JFree - Jasper	3.0.1 - 3.1.0	12	9	3	0	0
Mondrian - Jasper	1.3.4 - 2.0.0	3	3	0	0	0
Axis - JBoss	3.2.5 - 4.0.0	8	5	1	2	0
Hibernate - JBoss	4.2.0 - 4.2.1	29	25	0	3	1
JDO2 - Spring	2.0m1 - 2.0m2	8	8	0	0	0
JRuby - Spring	2.0.3 - 2.0.4	7	7	0	0	0

Table 6. Accuracy of i-Usage Operations Recommendation

Mine on	Adapt to	Usages	Rec.	$\surd$	Miss
3.2.5-3.2.8	3.2.5-4.0.5	6	4	4	2
4.0.5-4.2.3	4.0.5-5.0.1	26	25	25	1

7.2.3 Recommending Edit Operations for Adaptation

In this experiment, LIBSYNC was run on a development branch in JBoss' history to mine adaptation patterns for all libraries used by JBoss. We then ran LIBSYNC for adaptation recommendation on another branch which derives from the same branching point with the first branch but are independently developed onward. We manually checked the recommended operations against the actual adaptations in the second branch. A recommendation is considered correct if it has at least one correct operation at a correct location.

Table 6 shows the result. The first two columns show the development branches on which LIBSYNC mined the adaptation patterns and applied adaptation recommendations respectively. Column Usages shows the number of usage adaptations. Column Rec shows the numbers of recommended adaptations. As shown in Table 6, LIBSYNC provides highly correct recommendations. The recommended operations were correct as developers changed all of them except for three missing cases in which old usages were completely abandoned and totally new usages were used.

LIBSYNC was able to correctly recommend the adaptation for all examples in this paper. For example, LIBSYNC could recommend the correct adaptation for the case of JFreeChart in JBoss in Figure 1. This change happened in JBoss 3.2.8 in the branch from version 3.2.5 to 3.2.8 and was learned to adapt from version 4.0.1 to 4.0.2 in the branch from version 3.2.5 to 4.0.5. Those two changes were actually the patches to fix a bug of NullPointerException when using the deprecated constructor of DefaultTableXYDataset.

7.3 Adaptation of x-Usage

This section describes our evaluation of LIBSYNC in recommending the code locations for the adaptation of x-usages in JBoss. We used a wide range of versions in JBoss as described in Table 3. For each change in JBoss from version i to j, we used OAT to collect all changed APIs into the change set ΔL. We identified a set $XU(P, \Delta L_i)$, all methods in JBoss at version i that override some API's methods.

Each method in $XU(P, \Delta L_i)$ is considered for adaptation recommendation with the same operations as those operations that are detected on the overridden method in the API. A recommendation to a method at version i was con-

Table 7. Accuracy of x-Usage Recommendation

	Rec.	$\surd$	X
Name	6	4	2
Class name	1	1	0
Package name	2	2	0
Deprecated	3	3	0
Change parameter type	4	4	0
Del parameter	7	7	0
Change return type	6	6	0
Change exception	1	1	0
Add parameter-Change Exception	1	1	0
Add parameter-Change Return type	2	2	0

sidered correct if that method was really changed in the same way in the version j, otherwise, it was marked incorrect.

The result is shown in Table 7. Each row represents one particular type of changes in the external API(s). For example, the row Name is only for the methods with changed names. The row of Add parameter-Change Exception is for the methods changing in both parameter and exception that could be thrown. Therefore, the numbers in a column are exclusive from row to row. Column Rec shows the number of recommended locations. Columns $\surd$ and X respectively show the correctly and incorrectly detected locations for x-usage adaptation.

LIBSYNC provides highly correct recommendations. It is incorrect in only two cases out of the total of 33 cases. These two wrong cases have the same nature in which they are both caused by the incorrect mapping results from OAT when detecting the changes of the class PersistenceInfoImpl in javax API that is used in JBoss from version 4.0.3SP1 to 4.0.4GA. Instead of reporting two deleted methods (getPersistenceXmlFileUrl and setPersistenceXmlFileUrl), and two added methods (getPersistenceUnitRootUrl and setPersistenceUnitRootUrl), OAT reported two renaming operations. Therefore, the recommendation was two renaming operations while the correct adaptation should be two deletions and two additions. For other types of changes, the recommendations are all correct.

8. Related Work

This section describes related work on API evolution, API usage modeling, and adaptation.

8.1 Library Evolution and Client Adaptation

There are several existing approaches to support client adaptations to cope with evolving libraries. Chow and Notkin [7] proposed a method for changing client applications in response to library changes—a library maintainer annotates changed functions with rules that are used to generate tools that will update client applications. Henkel and Diwan's CatchUp [15] records and stores refactorings in an XML file that can be replayed to update client code. However, its update support is limited to three refactorings: renaming operations (e.g. types, methods, fields), moving operations (e.g. classes to different packages, static members), or change operations (e.g. types, signatures). The key idea of CatchUp, *record-and-replay*, assumes that the adaptation changes in client code are exact or similar to the changes in the library side. Thus, it works well for replaying rename or move refactorings or supporting API usage adaptations via inheritance. However, CatchUp cannot suggest programmers how to manipulate the context of API usages in client code such as the surrounding control structure, or the ordering between method-calls such as the example shown in Section 2. Moreover, CatchUp requires that library and client application developers use the same development environment to record API-level refactorings, limiting its adoption in practice.

SemDiff [8] mines API usage changes from other client code or the evolution of library itself, similar to our work. The key difference of LIBSYNC from SemDiff is that our work uses a graph-based representation to capture the context of an API usage, including the dependencies among method calls and with a surrounding control logic. In our work, an adaptation pattern is captured in term of a frequent set of graph editing operations that are common to multiple API usage skeletons before and after library migration. In contrast, SemDiff defines an adaptation pattern as a frequent *replacement* of a method invocation. That is, if a method call to $A.m$ is changed to $B.n$ in several adaptations, $B.n$ is likely to be a correct replacement for the calls to $A.m$. As SemDiff models API usages in terms of method calls, it cannot support complex adaptations that involve multiple objects and method calls and that require the knowledge of the surrounding context of those calls. LIBSYNC's key departure point is that when a library's API declarations are modified, such evolution often involves coordinating uses of multiple objects and multiple method calls under certain contexts.

Xing and Stroulia's Diff-CatchUp [43] automatically recognizes API changes of the reused framework and suggests plausible replacements to the obsolete APIs based on working examples of the framework codebase. Dig *et al.*'s MolhadoRef [11] uses recorded API-level refactorings to resolve merge conflicts that stem from refactorings; this technique can be used for adapting client applications in case of simple rename and move refactorings occurred in a library.

Tansey and Tilevich's approach [33] infers generalized transformation rules from given examples so that application developers use the inferred rules to refactor legacy applications. However, this approach focuses on annotation refactorings that replace the type and naming requirements to the annotation requirements of a target framework. Furthermore, this approach does not focus on updating client applications to cope with evolving libraries.

Andersen and Lawall [3] proposed *spdiff* that identifies common changes made in a set of files. API developers could use *spdiff* to extract a generic patch and apply it to other clients. Their approach models the changes at the level of *text-line* changes. On the other hand, LIBSYNC uses a graph-based representation to capture more thorough syntactic and semantic information for adapting API usages. SmPL [21,

27] is a domain-specific source transformation language that captures textual patches with a more semantic description of program changes. However, it does not explicitly distinguish API changes from their usage changes.

8.2 Program Differencing and Origin Analysis

Existing differencing techniques use similarities in names and structures to match code elements at a particular granularity: (1) lines and tokens [35], (2) abstract syntax tree nodes [13,23], (3) control flow graph nodes [5], (4) program dependence graph nodes [6], etc. For example, *diff* computes line-level differences per file using the longest common subsequence algorithm [17]. Our API usage comparison algorithm is similar to program differencing algorithms that it detects changes between two versions of an internal program representation using name-, content- and structure-based similarities. Zou and Godfrey [47] first developed an origin analysis technique to support software evolution analyses by mapping corresponding code elements between two program versions. Several other techniques [10,18,19,38,41, 47] improved and extended prior origin analysis techniques; some of these derive refactoring transformations—move a method, rename a class, add an input parameter, etc.—based on the matching result between two versions. OAT is similar to these techniques in that it maps corresponding code API declarations and API usage code fragments.

8.3 API Usage Specification Extraction

There exist several approaches for extracting API usage specifications. The forms of recovered API usage specifications and patterns include finite state automaton [37, 46], pairs of method calls [22, 39], partial orders of calls [1, 34], Computation Tree Logic formulas [36]. The API usage representations in those static approaches are still limited, for example, the patterns are without control structures and involve only individual objects belonging to one class. Our graph-based API usage representation captures multi-object API usage patterns with control structures.

In contrast to those static approaches, dynamic approaches recover the specifications by investigating the execution traces of programs [14, 28–30, 44]. These dynamic approaches require a huge amount of execution traces. Our graph-based representation, iGROUM, captures API usage patterns with control and data dependencies among method calls, and surrounding control logic such as `while` loop and `if` statement. The API usage representations in this paper extend our prior work on GrouMiner [25] to tailor the original multi-object usage representation in order to capture the relevant context surrounding the use of external APIs. In particular, iGROUM explicitly captures API types and methods that appear in action and data nodes, so that program slicing can isolate only a sub-graph that is relevant to the use of a particular library. We also created a new model called xGROUM to represent overriding and inheritance relationships between client methods and API methods.

8.4 Empirical Studies of API Evolution

Dig and Johnson [9] manually investigated API changes using the change logs and release notes to study the types of library-side updates that break compatibility with existing client code, and discovered that 80% of such changes are refactorings. Xing and Stroulia [42] used UMLDiff to study API evolution in several systems, and found that about 70% of structural changes are refactorings. Kim *et al.*'s signature change pattern analysis [20] categorizes API signature changes in terms of data-flow invariant. Yokomori *et al.* [45] investigated the impact of library evolution on client code applications using component ranking measurements. Padioleau *et al.* [26] found that API changes in the Linux kernel lead to subsequent changes on dependent drivers, and such collateral evolution could introduce bugs into previously mature code. These studies motivate the need for supporting complex client adaptations beyond replaying library-side refactorings in client code.

9. Conclusion and Future Work

This paper presents LIBSYNC that guides developers in adapting API usages in client code to cope with evolving libraries. LIBSYNC uses several graph-based techniques to recover the changes of API usage skeletons from codebase of other client systems, and recommends the locations and edit operations for adapting API usage code. The evaluation of LIBSYNC on real-world software systems shows that it is highly correct and useful. Especially, LIBSYNC can recover and recommend on complex API usage adaptations, which current state-of-the-art approaches are hardly able to support. One limitation of our approach is that it requires a set of programs that already migrated to a new library version under focus or adequate amount of API usages within the library itself. As it is not straightforward to identify which version of a library is used by client systems, we are currently in the process of developing a co-evolution analysis framework that can automatically extract the versioning information of libraries used by client systems in order to build a large corpus of API usage skeletons and to build a repository of API usage adaptation patterns.

References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC-FSE '07: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 25–34. ACM Press, 2007.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499. Morgan Kaufmann Publishers Inc., 1994.

[3] J. Andersen and J. Lawall. Generic patch inference. In *ASE ' 08: Proceedings of the 23rd IEEE/ACM International*

Conference on Automated Software Engineering, 2008, pages 337–346. IEEE Computer Society, 2008.

[4] A. Andoni and PiotrIndyk. E2 lsh 0.1 user manual. http://web.mit.edu/andoni/www/LSH/manual.pdf.

[5] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *ASE '04: Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pages 2–13. IEEE Computer Society, 2004.

[6] D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, 4(1):3–35, 1995.

[7] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 359. IEEE Computer Society, 1996.

[8] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 481–490. ACM Press, 2008.

[9] D. Dig and R. Johnson. The role of refactorings in API evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 389–398. IEEE Computer Society, 2005.

[10] D. Dig and R. Johnson. Automated detection of refactorings in evolving components. In *ECOOP '06: Proceedings of European Conference on Object-Oriented Programming*, pages 404–428. Springer, 2006.

[11] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 427–436. IEEE Computer Society, 2007.

[12] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 57–72. ACM Press, 2001.

[13] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling—tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):18, November 2007.

[14] M. Gabel and Z. Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 339–349. ACM Press, 2008.

[15] J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 274–283. ACM Press, 2005.

[16] J. W. Hunt and M. Mcilroy. An algorithm for differential file comparison. *Technical report*, 1976.

[17] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

[18] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 333–343. IEEE Computer Society, 2007.

[19] S. Kim, K. Pan, and J. E. James Whitehead. When functions change their names: Automatic detection of origin relationships. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 143–152. IEEE Computer Society, 2005.

[20] S. Kim, E. J. Whitehead, and . J. Bevan, Jr. Properties of signature change patterns. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 4–13. IEEE Computer Society, 2006.

[21] J. L. Lawall, G. Muller, and N. Palix. Enforcing the use of API functions in Linux code. In *ACP4IS '09: Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software*, pages 7–12. ACM Press, 2009.

[22] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *SIGSOFT Softw. Eng. Notes*, 30(5):296–305, 2005.

[23] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using Abstract Syntax Tree matching. In *MSR'05*, pages 2–6. ACM Press, 2005.

[24] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. In *FASE '09*, pages 440–455. Springer Verlag, 2009.

[25] 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 *Proceedings of the 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2009)*. ACM Press, 2009.

[26] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in linux device drivers. *SIGOPS Operating Systems Review*, 40(4):59–71, 2006.

[27] Y. Padioleau, J. L. Lawall, and G. Muller. SmPL: A domain-specific language for specifying collateral evolutions in linux device drivers. *Electronic Notes Theoretical Computer Science*, 166:47–62, 2007.

[28] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, pages 371–382. IEEE Computer Society, 2009.

[29] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *ICSE '07: Proceedings of 29th international conference on Software Engineering*, pages 240–250. IEEE CS, 2007.

[30] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *ISSTA '07: Proceedings of the international symposium on Software testing and analysis*, pages 174–184. ACM Press, 2007.

[31] JBoss: An open source, standards-compliant, J2EE based application server. http://sourceforge.net/projects/jboss/.

[32] Subversion.tigris.org. http://subversion.tigris.org/.

[33] W. Tansey and E. Tilevich. Annotation refactoring: inferring upgrade transformations for legacy applications. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 295–312. ACM Press, 2008.

[34] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proc. 24th International Conference on Automated Software Engineering (ASE 2009)*, pages 283–294. IEEE Computer Society, 2009.

[35] W. F. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, 1984.

[36] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, pages 295–306. IEEE Computer Society, 2009.

[37] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *ESEC-FSE '07: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on Foundations of software engineering*, pages 35–44. ACM Press, 2007.

[38] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 231–240. IEEE Computer Society, 2006.

[39] C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Trans. Softw. Eng.*, 31(6):466–480, 2005.

[40] W. Wu, Y.-G. Gueheneuc, G. Antoniol, and M. Kim. Aura: A hybrid approach to identify framework evolution. In *ICSE '10: Proceedings of the 32nd International Conference on Software Engineering*. ACM Press, 2010.

[41] Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65. ACM Press, 2005.

[42] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported - an Eclipse case study. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 458–468. IEEE Computer Society, 2006.

[43] Z. Xing and E. Stroulia. API-evolution support with Diff-Catchup. *IEEE Trans. Softw. Eng.*, 33(12):818–836, 2007.

[44] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 282–291. ACM Press, 2006.

[45] R. Yokomori, H. Siy, M. Noro, and K. Inoue. Assessing the impact of framework changes using component ranking. In *Proceedings of the International Conference on Software Maintenance*, pages 189–198. IEEE Computer Society, 2009.

[46] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, ACM Press, 2009.

[47] L. Zou and M. W. Godfrey. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.