A Graph-based Approach to API Usage Adaptation

Authors: Nguyen, Hoan Anh, et al

Presenter: Zongtao Lin

Background

Application Programming Interfaces (API) is an interface that defines interactions between multiple software components and resources. It defines the kinds of calls or requests that can be made, how to make them, the data formats that should be used, the conventions to follow, etc.

API can be updated or changed by declaring new attributes, accommodating the internal logic, etc.

Problem statement

Reusing existing library components can highly reduce cost of software development and maintenance. **However**, library components evolve might lead to complex API change. It might lead to existing client break. **Thus**, the client's codes are also need to be changed to corresponding the library change.

```
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

Existing limitation

Some existing research techniques require library maintainers and client application developers to use the same development environment to record and replay refactorings. → **ideal case, not happen commonly**

Existing API usage modeling and extraction techniques are limited by simplified representations such as a sequence of method calls. → **Cannot handle complex control and data dependencies of API usage**

Proposing a set of graph-based models and algorithms to capture updates in evolving libraries and updates in client applications associated with changes in the libraries.

Another algorithm to summarize edit operations from API usage code samples before and after library migration.

Approach

Four main approaches:

1. **Origin Analysis Tool (OAT):** A tree-based origin analysis technique maps corresponding code elements between two versions.
2. **Client API Usage Extractor (CUE):** A graph-based representation that extracts API usage skeletons from client code and the use of APIs within the library.
3. **Usage Adaptation Miner (SAM):** A graph alignment algorithm can identify API usage changes and mine the adaptation patterns from a set of API usages by another algorithm.
4. **LIBSYNC:** A tool provide recommend locations and edit operations for adapting API usage code in the client.

Motivation Example

JBoss is a large Java project. It has about 40,000 methods and uses up to 262 different libraries. If one external API library were changed, JBoss might need to have some API usage adaptations. The approach author proposed could save a lot of time for JBoss maintenance.

The API usage adaption is mainly composed with two types: **invocations** and **inheritance**.

API Usage via Method Invocations

added code

```
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

```
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

~~deleted code~~

API Usage via Method Inheritance

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

Object-oriented programming pattern

Object-oriented programming has two common ways to use the API functionality

- **Method invocation:** directly calling to API methods or creating objects of API classes
- **Inheritance:** declaring classes in client code that inherit from the API classes and override their methods

Based on those patterns, API usage adaptation model can capture complex context surrounding API usages by

- Data and ordering dependencies among API usages
- Control structures around API usages
- Interaction among multiple objects of different types

Origin Analysis Tool (OAT)

The purposes of OAT: 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

OAT views a program P (either a library or client) as a project tree T(P) which has three attributes:

- Declaration (declare(u))
- Parent (parent(u))
- Content (content(u)):

Origin Analysis Tool (OAT) — Similarity

How to measure the similarity between two nodes?

Declaration attribute similarity s_d

Content attribute similarity s_c

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

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

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

Origin Analysis Tool (OAT) — Mapping Algorithm

OAT classified each node into three categories.

- AM nodes, node already mapped to another node
- PM nodes, its parent node is mapped but it is not mapped to any node
- UM nodes, the node and its parent are not mapped

After a UM node were mapped with another node, both node will classified into AM nodes and their sub nodes will become PM nodes.

```
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
```

Client API Usage Extractor (CUE) — i-Usage Model

CUE represents the API i-usages in clients via a graph-based model called iGROUM (invocation-based, GRaph-based Object Usage Model)

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.*

Client API Usage Extractor (CUE) — i-Usage Model

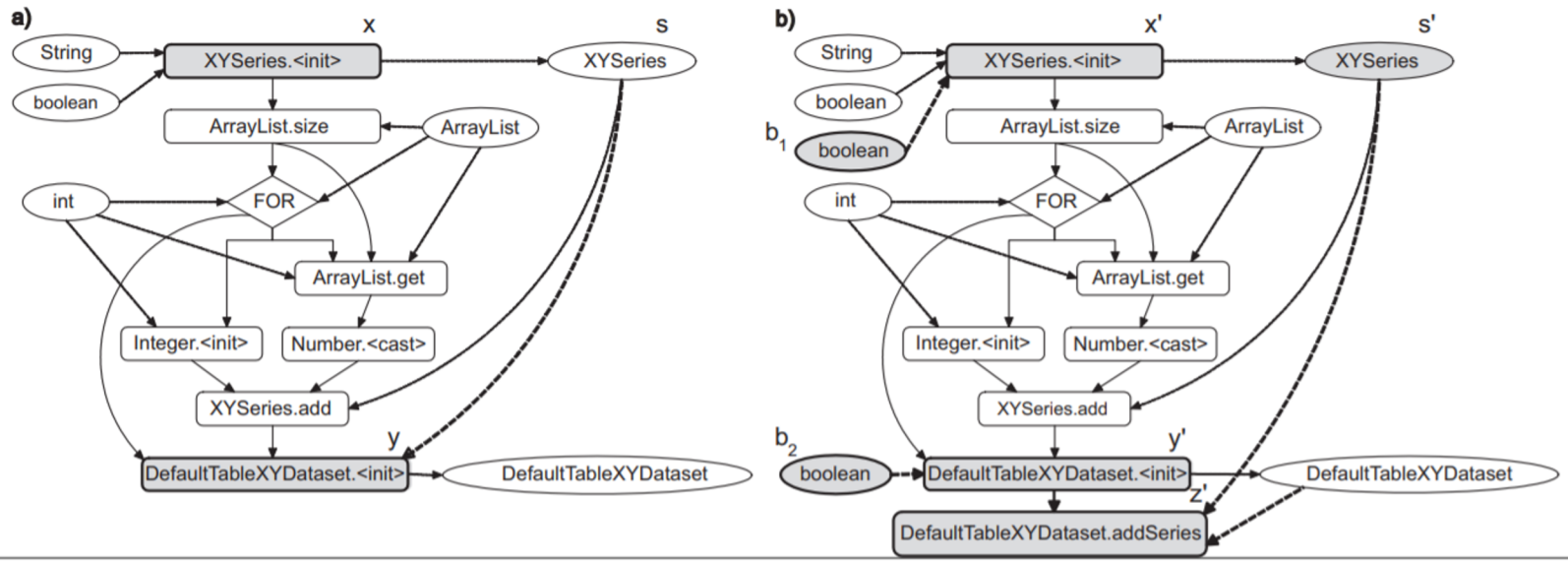

Figure 7. API i-Usage models in JBoss before and after migration to a new JFreeChart library version

Client API Usage Extractor (CUE) — x-Usage Model

CUE use 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

There are two kinds of edges: o-edge (overriding) and i-edge (inheriting)

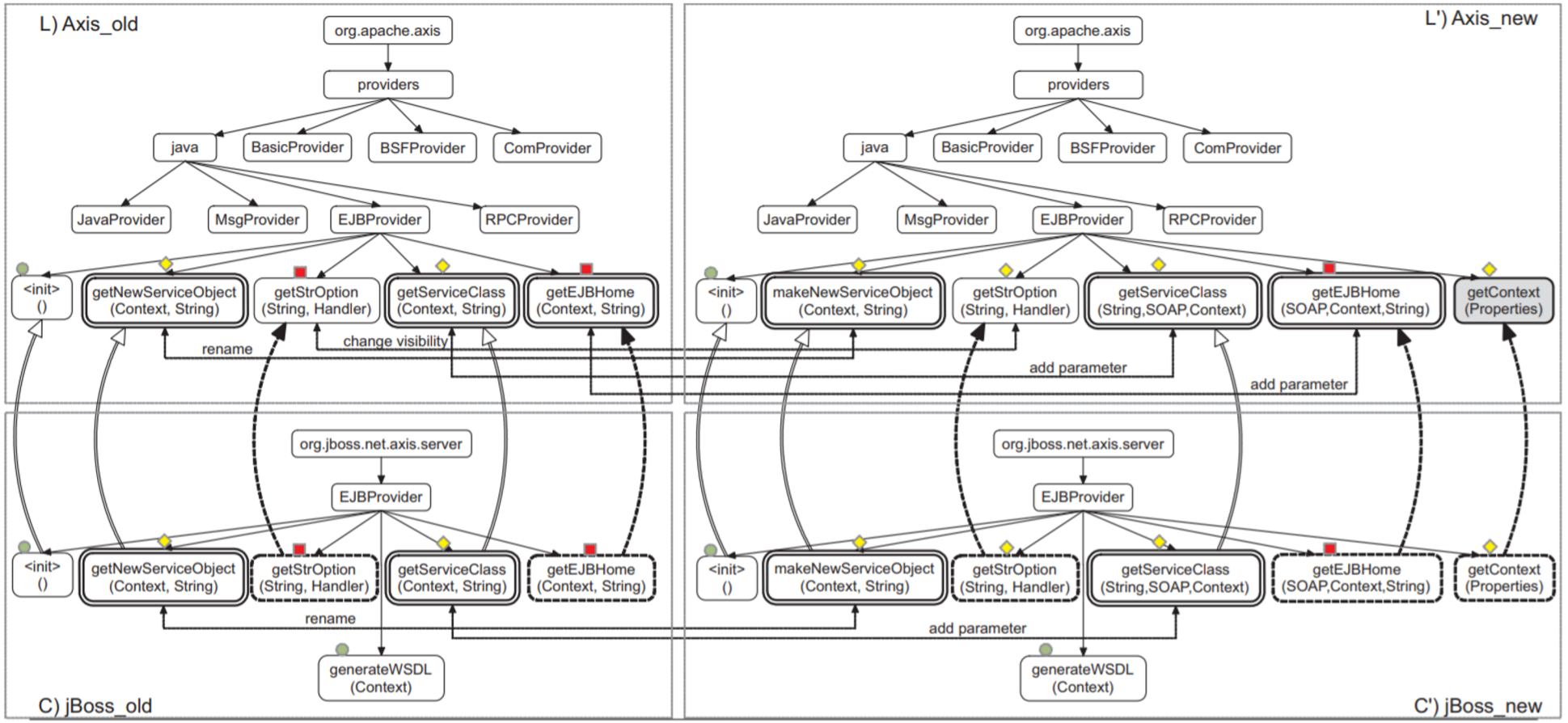

Figure 8. API x-Usage models in JBoss before and after migration to a new Axis library version

Usage Adaptation Miner (SAM)

SAM uses iGROUMs to represent API i-usages in client codes. SAM also uses a graph alignment algorithm to identify API i-usage changes, and then use a mining algorithm to find API usage adaptation patterns.

```
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

Usage Adaptation Miner (SAM)

```
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

Recommending Adaptations (LIBSYNC)

Locate the recommendation location by Client API update extractor (CUE).

Based on iGROUM or xGROUM find the i-usage or x-usage recommendation pattern.

Those patterns were mined by Usage Adaptation Miner(SAM).

Finally, provide suggestion on exact location with mined pattern.

Experiment Evaluation

Quality of change detection in OAT: Precision = # of correctly detected / # of total

TP = # of correct pairs / # of total

FP = # of incorrect pairs / # of total

Table 1. Precision of Origin Analysis Tool OAT

Version Pairs	Mapped	Checked	√	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%

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%

Experiment Evaluation

Use precision to evaluate performance of i-Usage changes detection

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

Experiment Evaluation

Table 5. Accuracy of i-Usage Location Recommendation

API - Client	Version	Rec.	√	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

Related Work

The author made a lot of comparisons with existing similar method in each stage.

The author's approaches is different from others because it use graph-based representation to extracts API usage which increase the accuracy of API usage adaptation. The author use two metrics to calculate similarity in OAT which outperform than others.

Conclusion

The author present tool LIBSYNC which can adapting API usages in client code along with libraries evolution. LIBSYNC uses several graph-based techniques and tree-based techniques to increase its accuracy and precision.

By comparing LIBSYNC with other existing techniques, LIBSYNC performs better than others in overall perspective.

Discussion

- Any flaws on author’s approaches?
- Will you use this tools on programming?
- Any other new approaches for API usage adaption?

Thank you !