2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation

ACUA: API Change and Usage Auditor

Wei Wu, Bram Adams, Yann-Gaël Guéhéneuc, Giuliano Antoniol
DGIGL, École Polytechnique de Montréal
Montréal, Canada
Email: {wei.wu, bram.adams, yann-gael.gueheneuc, giuliano.antoniol}@polymtl.ca

***Abstract*—Modern software uses frameworks through their Application Programming Interfaces (APIs). Framework APIs may change while frameworks evolve. Client programs have to upgrade to new releases of frameworks if security vulnerabilities are discovered in the used releases. Patching security vulnerabilities can be delayed by non-security-related API changes when the frameworks used by client programs are not up to date. Keeping frameworks updated can reduce the reaction time to patch security leaks. Client program upgrades are not cost-free, developers need to understand the API usages in client programs and API changes between framework releases before conduct upgrading tasks. In this paper, we propose a tool ACUA to generate reports containing detailed API change and usage information by analyzing the binary code of both frameworks and clients programs written in Java. Developers can use the API change and usage reports generated by ACUA to estimate the work load and decide when to starting upgrading client programs based on the estimation.**

I. Introduction

Developers often use frameworks in software development today and framework APIs may change between releases [1]. Client programs need to upgrade to use new features or patch security vulnerabilities. Upgrading frameworks is not cost-free and may interrupt the services. For example, the online tax filing service of Canada Revenue Agency was down for five days to patch the Heartbleed bug[1]. The new releases fixing security vulnerabilities of frameworks should not contain API changes, but in practice, framework providers usually do not patch all affected releases, but the latest one. Therefore, if the version of the framework used in client programs is too old, the upgrading process can be slow down by adapting API changes that are unrelated to the vulnerabilities. Hence, keeping frameworks updated can reduce the reaction time to patch security leaks.

To keep frameworks updated, developers have to estimate the cost each framework upgrade before initiating any action. They need the following information to plan upgrading tasks: (i) what, where and how are the APIs of frameworks used in client programs? (ii) which of the used APIs are changed in the new releases of the frameworks and how are they changed?

The answers to the two questions influence the upgrading cost in client programs. A changed API used in many locations in client programs may cause Shotgun Surgery [2], a code smell causing lots of little changes in lots of classes, which would result in high upgrade costs. Different types of API changes affect upgrade costs differently. For example, the addition of a formal parameter to an API method is easier to adapt than the removal of an API method. To adapt the latter change, developers must find a replacement of the removed method or re-implement it, while for the first case, they can simply provide a new parameter.

To assist development teams in assessing the cost of framework upgrades, we develop ACUA (API Change and Usage Auditor), a tool to collect API change and usage information through the analysis of the binary code of frameworks and client programs. For a given client release and a pair of framework releases, ACUA first reports which (and where) APIs are used in the client program. Second, ACUA detects what (and how) APIs are changed between the currently-used and the to-be-upgraded releases of frameworks. Third, ACUA verifies which of the changed APIs affect the client program.

To the best of our knowledge, ACUA is the first tool that analyzes frameworks and their client programs and provides both API change and usage information with detailed classifications. Previous tools either report API changes between framework releases or API usages of a specific framework release. Yet, tools like ACUA would help developers plan and act efficiently regarding API change adapting. For the developers of client programs, a good knowledge of how APIs are used and how they are affected by different types of API changes would help them better estimate upgrading work load and plan framework upgrades. For framework developers, such tools would help them document the API changes that are difficult to adapt in high priority. To demonstrate how to use ACUA, we use Apache solr-core[2] v3.6.2 and v4.0.0 as a running example.

The remainder of the paper is organized as follows. Section II introduces background information about API change and usage. Section III describe the functionality of ACUA. Section IV discusses the deference between ACUA and compilers. Section V summarizes the related works. Finally, Section VI concludes the paper and outlines future works.

II. Background

This section presents some background information about APIs changes and usages.

A. API changes

Frameworks provide their services through APIs. These APIs may change during the evolution of frameworks. Des Rivières [3] has summarized API changes, based on entities changed in APIs, *e.g.,* packages, classes, modifiers *etc.*. He also pointed out which API changes may cause binary incompatibility, *i.e.*, the class files of the new releases of frameworks cannot be linked to client programs without recompilation. Although an API change may only cause binary

[1]http://www.cra-arc.gc.ca/gncy/fq-hb-eng.html

[2]http://lucene.apache.org/solr/

978-1-4799-6148-1/14 $31.00 © 2014 IEEE
DOI 10.1109/SCAM.2014.33

TABLE I. FRAMEWORK API CHANGE TYPES - REFERENCE TYPE LEVEL

ACUA	Des Rivières
Non-Existing Class (NEC)	Delete API type from API package
Non-Existing Interface (NEI)	
Moved Class (MC)	
Moved Interface (MI)	
Decrease Access (DA)	Change public type in API package to make non-public
Change Type Kind (CTK)	Change kind of API type (class, interface, enum, or annotation type)
Expand Super Interface Set (ESIS)	Expand superinterface set (direct or inherited)
Contract Super Interface Set (CSIS)	Contract superinterface set (direct or inherited)
Add Method To Interface (AMTI)	Add API method to Interface
Add Abstract Method (AAM)	Add Abstract API method to class
Contract Super Class Set (CSCS)	Contract superclass set (direct or inherited)
Change To Abstract (CTA)	Change non-abstract to abstract
ChangeToFinal (CTF)	Change non-final to final
Change Type Bound (CTB)	Add, delete, or change type bounds of type parameter
	Add type parameter
	Delete type parameter
	Re-order type parameters
With Changed Method (WCM)	Add API method
	Delete API method
	Move API method up type hierarchy
	Move API method down type hierarchy
	All method level changes

TABLE II. FRAMEWORK API CHANGE TYPES - METHOD LEVEL

ACUA	Des Rivieres
Non-Existing Method (NEM)	Change method name
	Delete API Method
	Move API method up type hierarchy
	Move API method down type hierarchy
Change Formal Parameter (CFP)	Add or delete formal parameter Change type of a formal parameter
Change Return Type (CRT)	Change result type (including void)
Decrease Method Access (DMA)	Decrease access: from public access to protected, default, or private access from protected access to default or private access;
Change Method To Abstract (CMTA)	Change non-abstract to abstract
Change Method To Final (CMTF)	Change non-final to final
Change Method To Static (CMTS)	Change static to non-static
Change Method To Non Static (CMTNS)	Change non-static to static

or source incompatibility [4], most of the API changes listed by Des Rivière also cause source incompatibility, *i.e.*, there are errors when the client program source code are compiled with the new releases of frameworks. In this study, we do not distinguish between these two types of incompatibilities because, as stated by Buchholz, [5] *"Every change is an incompatible change" (a risk/benefit analysis is always required)* and hence client developers should be informed of all API changes causing both source or binary incompatibilities.

In the classification of API changes proposed by Des Rivières, we selected those related to classes, interfaces and methods, because they are the fundamental entities of object-oriented programming languages. We summarize Des Rivières' API changes into 23 categories, among which, 15 are at reference type level (shown in Table I) and 8 are at method level (shown in Table II).

These types of API changes do not have the same effect on client programs; some of them are more difficult to adapt than the others. For example, the addition of a new `int` parameter to an API method is easier to adapt than the removal of an API method. To adapt the latter change, developers must find a replacement of the removed method or re-implement it. Knowing the types of API changes is important for accurate estimations of programs upgrade workloads.

B. API Usages

To access framework APIs, client developers may extend a class, implement an interface, use framework reference types (or their subtypes as generic types, method return types or formal parameter types), or call methods defined in frameworks. These different forms of APIs usages have an impact on the process of adapting client programs to new releases of frameworks. Class extensions and interface implementations are two inheritance-style usages that require a good knowledge of the internal implementation of frameworks and API changes impact more client programs [6]. On the one hand, it is not possible to eliminate API inheritance-style usages completely. Frameworks are designed for the purpose of inversion of control (IOC) [7], *i.e.*, client programs become a part of frameworks by overriding methods in classes or interfaces provided by the frameworks. On the other hand, framework classes not designed for IOC can be replaced by composition-style usage.

Composition-style usages encapsulate framework APIs and use local APIs to hide them, while inheritance-style usages retain framework APIs. An example of composition-style usage is the use of framework reference types as private members in client classes while only accessing them within method bodies.

In general, API changes in inheritance-style usage are more difficult to adapt because they require a good knowledge of the internal implementation of frameworks. Inheritance-style usages are not avoidable in client programs for the purpose of IOC, but those not for IOC can be converted to composition-style usage.

III. ACUA

In this section, we first present the modules of ACUA in brief. Then, we describe its API change and usage detection algorithms. Last, we demonstrate the working flow, the inputs and the outputs of ACUA with an running example.

A. Modules

The function modules of ACUA are shown in Figure 1. The elements with gray background are the modules of ACUA and the those with white background are the inputs and outputs of each module.

ACUA takes two versions of Maven POM (Project Object Model) configuration files of client programs as inputs. Maven is a project management tool from the Apache Software Foundation[3]. Maven projects store the information about the dependencies between client programs and frameworks in POM files. Framework Usage Analyzer (FUA) uses the information in POM files to detect framework version changes. FUA also requires a connection to remote or local Maven repository to download the jar files of the corresponding versions of the frameworks and the client program. FUA interacts with Maven repository through Aether library[4].

The advantage of using Maven POM files is that Maven is widely used in software development and provides the complete information required by ACUA, such as dependencies and jar file locations. Non-Maven programs need to be converted to Maven projects and analyzed by ACUA. The conversion can be done automatically if the program dependency information is explicitly described, such as in Eclipse plug-ins, because ACUA has a model to represent program evolution and dependencies. For programs whose dependencies do not always have version information, like projects managed by Apache Ant, developers must complete the version information in a semi-automatic way or manually.

[3] http://maven.apache.org/

[4] http://www.eclipse.org/aether/

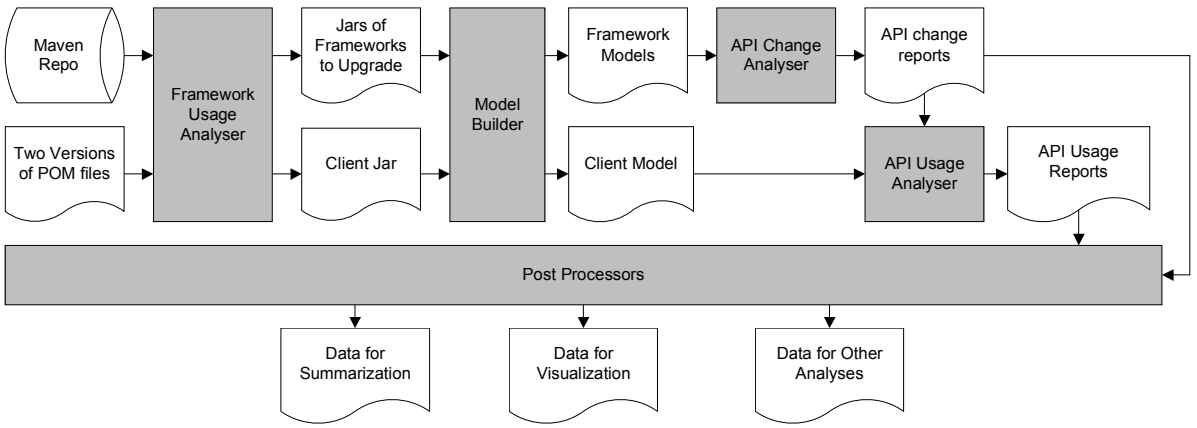


Fig. 1. ACUA Modules

Next, Model Builder parses the jar files of the frameworks and the client program to build the models containing reference types, method definitions, call and inheritance graphs of framework and client program releases, according to our meta-model. Model builder uses the ASM Java bytecode analysis framework[5] to extract the model data from the jar files.

Taking the models as input, API Change Analyzer (ACA) detects the API changes between the two releases of frameworks and classifies them into the types presented in Section II. ACA outputs the classified API changes as API change reports of the two releases of each framework.

Based on the API change reports and the client model, API Usage Analyzer (AUA) detects where and how the APIs of the frameworks used in the client program, then verifies if the used APIs are changed in the new releases of frameworks and affected by which types of API changes. As the output, AUA generates API usage reports for each framework including how the used APIs are affected by API changes.

B. API change analysis

Here, we present the algorithm to detect API changes between two releases of a framework. ACA detects API changes at reference type (classes, interfaces etc.) level and method level.

To analyze reference type level API changes, ACA first classifies the reference types of each pair of framework releases into four categories: Stable in old release, Changed in old release, Stable in new release, Changed in new release. Here, *Stable* stands for existing in both releases and *Changed* means existing only in one release, according to their package names and reference type names, without considering type kind (class or interface), modifiers, and generic type parameters. So, Stable reference types in both releases may have differences in their methods and Changed reference types may have counterparts with the same methods.

Then, it checks if the changed types in old release have counterparts in the new release with the same type names, but different package names. Those having a counterpart are classified as Moved types (classes, interfaces etc.) and the rest are classified as Non-existing types.

[5]asm.ow2.org

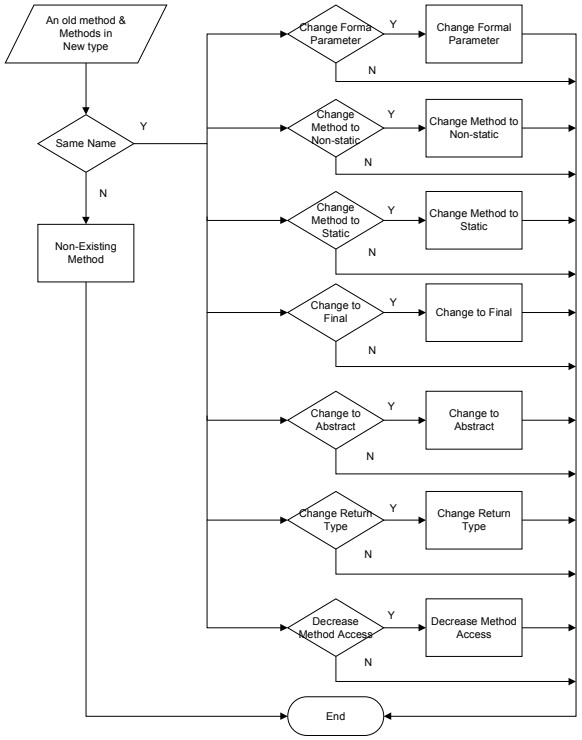


Fig. 3. Method-level API change detection algorithm

Next, ACA checks for stable types between the two releases and detect the types of API changes other than Non-Existing Classes and Interfaces. Figure 2 is the flowchart of the reference type level API change type detection algorithm.

For the stable reference types between two releases, ACA detects method level API changes as follows: First, ACA checks if there is another method with the same name in the old release of the framework. If there is not, ACA classifies the method as a Non-Existing Method. If there is one, ACA checks if the method level API changes other than Non-Existing Method happened to the method. The flowchart of the method level detection algorithm is shown in Figure 3.

A changed API can be tagged with more than one change types. For example, added field to interface and added method

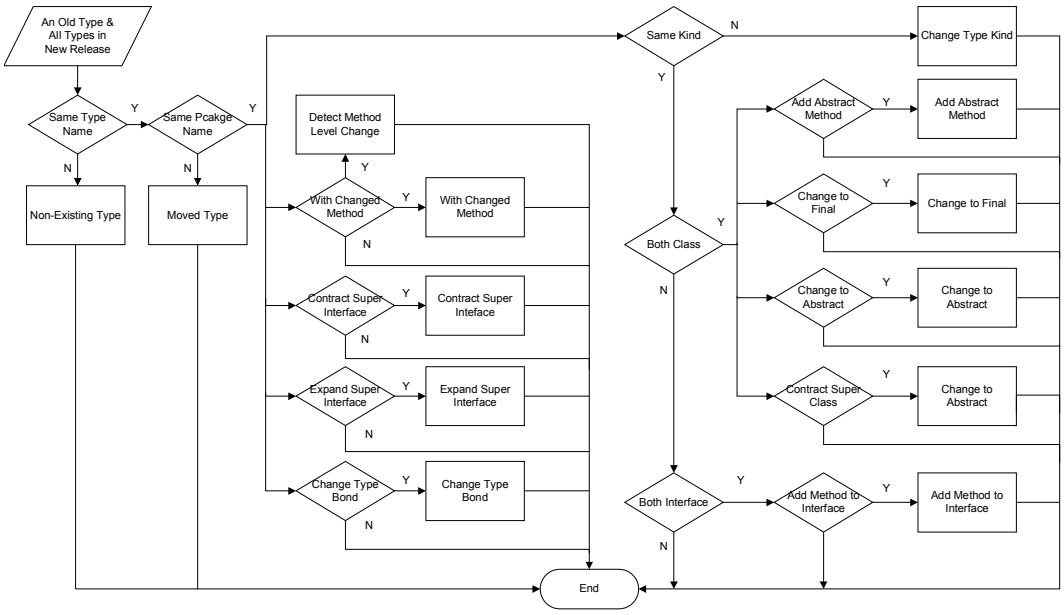


Fig. 2. Type-level API change detection algorithm

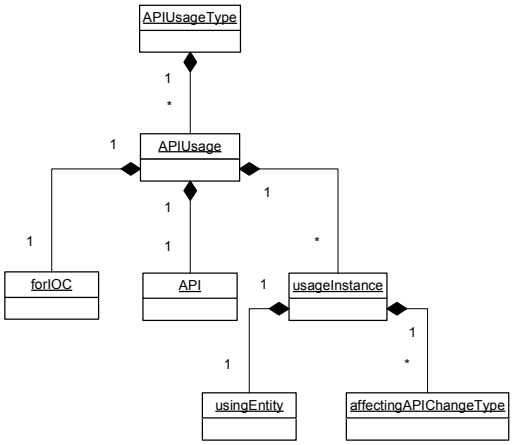


Fig. 4. API usage information

to interface can happen to the same interface.

C. API Usage Analysis

AUA checks which (and how) APIs are used by analyzing APIs call and inheritance graphs. It reports the following usage types: extensions of framework classes, implementations of framework interfaces, overrides of framework methods, and invocations of framework methods. For each usage type, as shown in Figure 4, AUA collects which APIs are used, if they are used for inversion of control, where they are used and the types of API changes by which they are influenced.

D. Working flow

Here, we describe the working flow of ACUA using solr-core[2] version 3.6.2 as a running example. Solr is a open source search platform based on Apache Lucene[6]. One framework

[6]http://lucene.apache.org/

```
...
<groupId>org.apache.solr</groupId>
<artifactId>solr-core</artifactId>
<version>4.0.0</version>
...
<dependencies>
        <dependency>
                <groupId>org.apache.lucene</groupId>
                <artifactId>lucene-core</artifactId>
                <version>4.0.0</version>
        </dependency>
...
```

Fig. 5. Snippet of solr-core v4.0.0 POM file

used by solr-core v3.6.2 is lucene-core v3.6.2 and the Solr development team wants to upgrade to lucene-core v4.0.0 in the next release (assuming it is also v4.0.0). They use ACUA to collect the information about the API changes between lucene-core v3.6.2 and v4.0.0 and how the changes used in Solr v3.6.2, because they need to estimate the upgrading work load and plan upgrading task accordingly.

Solr developers first create a POM file for v4.0.0 in which the version of lucene-core is changed to v4.0.0 according to the upgrading goal. A snippet of the POM fileis shown in Figure 5. In POM file, The releases of solr-core and lucene-core are represented by `<groupId>`, `<artifactId>`, and `<version>`. The release information of lucene-core is under `<dependency>` node and that of solr-core is at the root level.

Then, Solr developers run ACUA with the two POM files of Solr, v3.6.2 and v4.0.0 respectively. ACUA detects the version changes in lucene-core (v3.6.2 to v4.0.0), then it analyzes the jar files of the two releases of lucene-core and solr-core v3.6.2 to generate API change and API usage reports for them. To facilitate post processing, API change and API usage reports generated by ACUA are in XML format. Figure 6 is a part of API change report for lucene-core v3.6.2 to v4.0.0. In the report, the information of two releases of frameworks, the types of API changes, the levels and the changed APIs are stored

```
<Incompatibilities>
    <programName>
     org.apache.lucene:lucene-core
    </programName>
    <oldVersion>3.6.2</oldVersion>
    <newVersion>4.0.0</newVersion>
    <incompatibilities>
        <type>AddAbstractMethod</type>
        <instances>
            <level>type</level>
            <oldAPI>
             class org.apache.lucene.analysis.Analyzer
            </oldAPI>
        </instances>
                ...
    </incompatibilities>
        ...
</Incompatibilities>
```

Fig. 6. Snippet of lucene-core v3.6.2-v4.0.0 API change report

```
<exposureReport>
    <clientName>
     org.apache.solr:solr-core
    </clientName>
    <clientVersion>3.6.2</clientVersion>
    <frameworkName>
     org.apache.lucene:lucene-core
    </frameworkName>
    <oldVersion>3.6.2</oldVersion>
    <newVersion>4.0.0</newVersion>
    <apiUsages>
        <type>EXTENSION</type>
        <apiUsages>
            <oldAPI>
             class org.apache.lucene.analysis.Analyzer
            </oldAPI>
            <invocatedInFramework>
             true
            </invocatedInFramework>
            <instances>
                <entities>
                 class org.apache.solr.analysis.TokenizerChain
                </entities>
                <incompatibilityType>
                 AddAbstractMethod
                </incompatibilityType>
            </instances>
             ...
        </apiUsages>
                ...
    </apiUsages>

        ...
</exposureReport>
```

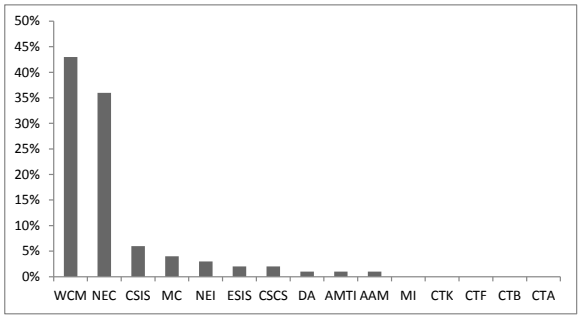
Fig. 7. Partial solr-core v3.6.2 API usage report on lucene-core v3.6.2-v4.0.0

in specific nodes. Developers can easily search, analyze them according to the information that they are interested in.

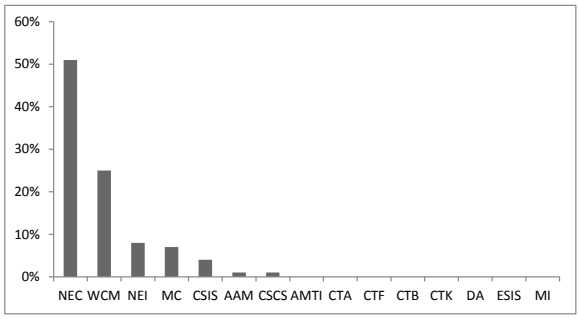
In a similar way, the API usage report (shown in Figure 7) lists the information of the release of the client program and of the two releases of the framework to upgrade, the used APIs, the types of the usages and the change types affecting the APIs in the new release of the framework.

The reports in XML format are easy to automatically process but difficult to read by human. ACUA also provides post-process modules to convert the reports to plain text format and to summarize the contents in the reports for statistical analysis or visualization. For example, developers can process the API change and API usage reports and generate the graph of the distributions of the types of API changes in the frameworks and compare it with that of the distributions of the types of API changes used in the client programs.

In Figure 8, we can see that WithChangedMethod is the most often API change type in the frameworks used by solr-core. It affects solr-core less often than NonExistingClass. Finding the replacements of the NonExistingClass could be



(a) API changes in frameworks



(b) API changes used in solr-core

Fig. 8. Reference type level API changes between the framework releases used by solr-core v3.6.2 and v4.0.0

time-consuming and Solr development team should take it into consideration when estimate the upgrading work load.

ACUA and its complete analysis results on solr-core v3.6.2 and v4.0.0. are available on our Web page[7].

IV. Discussion

Compilers are basic tools that report API change impacts in client programs and developers still need them while conducting concrete upgrading tasks to verify changes and generate binary code. However, compilers cannot help developers much to make upgrading plan and estimate upgrading workloads. Traditional compilers do not provide the following information that ACUA reports:

a) API change type:: compilers can detect API changes by showing compiling errors, but they do not report which types of API changes caused the errors. Because API changes are not equally difficult to adapt, such information is useful to plan program upgrade effectively. ACUA analyze client programs and two releases of frameworks to generate API usage report that summarize which APIs are used , where they are used (in client programs) and the types of API changes affecting them. With API usage reports, developers have a clearer picture of API change impacts.

b) API infiltration:: compilers cannot report how widely APIs are used in client programs. APIs keep evolving, developers should know how seriously APIs infiltrate, *i.e.*, are used in different locations, in their client programs to prepare for future API changes. If large numbers of the APIs of a framework widely used in client programs, it will be difficult to adapt to major changes in the frameworks or to switch to

[7] http://ptidej.net/downloads/replications/scam2014tools

other frameworks with similar functions. The API usage data collected by ACUA can be used to visualize API infiltration.

V. Related Work

Many existing tools help developers on framework API changes and usages. CatchUP! [8] records the refactoring operations in one release and replay them in another. AURA [9], Diff-CatchUp [10], HiMa [11], and SemDiff [12] generate the API change rules between releases of frameworks. Twinning [13] adapts different Java frameworks with similar functionalities. MAN [14] maps APIs between Java and C#. Portfolio [15] searches and visualizes relevant functions and their usages from a database. LibSync [16] helps developers learn complex API usage change patterns from the clients that have been already upgraded. Change Distilling [17] rebuilds change roadmaps between releases of programs. LSdiff [18] summarizes the changes between releases of frameworks into systematic structural differences and presents anomalies in them. Ref-Finder [19] detects the 63 types of refactoring. MADMatch [20] matches evolving program elements with an approach that uses Error Tolerant Graph Matching algorithm. Exapus [21] explores API usages from different views. APIMiner [22] extracts API usage examples using program slicing technique. Web API is a new type of APIs emerged with Internet. Espinha *et al.* [23] studied the evolution of four popular Web APIs and their impacts on clients. The previous tools only focus on one aspect of framework APIs, either on API changes or on API usages. To the best of our knowledge, ACUA is the first tool to report both API changes and usages in frameworks and client programs and classify them in detailed categories which can help developers estimate upgrading cost more accurately.

VI. Conclusion

When frameworks evolve, APIs may change between releases. Nowadays, patching security vulnerabilities of programs becomes an important reason to upgrade frameworks and should be done as soon as possible. Usually, framework providers only fix security leaks in the latest releases. If the releases of frameworks used are much behind , applying security patches may be delayed by adapting non-security-related API changes. Therefore, keeping framework updated should be considered as a regular task in software maintenance. In this paper, we propose a tool ACUA to generate API change and usage reports by analyzing the source or binary code of both frameworks and clients programs in Java. These reports help developers plan proactive framework upgrading. To assess the benefits of ACUA, we conduct a case study with 11 framework releases and their client program releases from a framework ecosystem, Eclipse. With the reports generated by ACUA, developers can know how API are used in and which API changes affect their client programs. Such information can help them to estimate upgrading work load and plan upgrading tasks. Empirical studies to quantitatively evaluate the benefits of ACUA and tools using the reports generated by ACUA to directly help framework upgrading are future works.

Acknowledgments

We thank Daniel German for providing the Maven central repository snapshots. This work has been partly funded by the NSERC Research Chairs on Software Patterns and Patterns of Software and on Software Change and Evolution.

References

[1] S. Raemaekers, A. van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *ICSM*, 2012, pp. 378–387.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[3] D. Rivières, "Evolving java-based apis 2," 2008. [Online]. Available: http://wiki.eclipse.org/Evolving_Java-based_APIs_2

[4] J. Dietrich, K. Jezek, and P. Brada, "Broken promises: An empirical study into evolution problems in java programs caused by library upgrades," in *CSMR-WCRE*, 2014, pp. 64–73.

[5] M. Buchholz, "Kinds of compatibility: Source, binary, and behavioral," 2008. [Online]. Available: https://blogs.oracle.com/darcy/entry/kinds_of_compatibility

[6] J. Bloch, *Effective Java (2nd Edition) (The Java Series)*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] J. Henkel and A. Diwan, "Catchup!: capturing and replaying refactorings to support api evolution," in *ICSE*, 2005, pp. 274–283.

[9] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *ICSE*, 2010, pp. 325–334.

[10] Z. Xing and E. Stroulia, "API-evolution support with diff-CatchUp," *IEEE TSE*, vol. 33, no. 12, pp. 818 – 836, December 2007.

[11] S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in *ICSE*, 2012, pp. 353–363.

[12] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *TOSEM*, vol. 20, no. 4, pp. 19:1–19:35, Sep. 2011.

[13] M. Nita and D. Notkin, "Using twinning to adapt programs to alternative apis," in *ICSE*, 2010, pp. 205–214.

[14] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining api mapping for language migration," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ser. ICSE '10, 2010, pp. 195–204.

[15] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceeding of the 33rd international conference on Software engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 111–120.

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

[17] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, vol. 33, pp. 725–743, November 2007.

[18] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 309–319.

[19] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: a refactoring reconstruction tool based on logic query templates," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, ser. FSE '10, 2010.

[20] S. Kpodjedo, F. Ricca, P. Galinier, G. Antoniol, and Y.-G. Guéhéneuc, "Madmatch: Many-to-many approximate diagram matching for design comparison," *IEEE TSE*, vol. 39, no. 8, pp. 1090–1111, 2013.

[21] C. D. Roover, R. Lammel, and E. Pek, "Multi-dimensional exploration of api usage," in *ICPC*, 2013, pp. 152–161.

[22] J. E. Montandon, H. Borges, D. Felix, and M. T. Valente, "Documenting apis with examples: Lessons learned with the apiminer platform," in *WCRE*. IEEE, 2013, pp. 401–408.

[23] T. Espinha, A. Zaidman, and H.-G. Gross, "Web api growing pains: Stories from client developers and their code," in *CSMR-WCRE*, 2014, pp. 84–93.