2011 19th IEEE International Conference on Program Comprehension

Modeling Framework API Evolution as a Multi-Objective Optimization Problem

Wei Wu
DGIGL, École Polytechnique de Montréal, Canada
Email: wei.wu@polymtl.ca

***Abstract*—Today's software development depends greatly on frameworks and libraries. When their APIs evolve, developers must update their programs accordingly. Existing approaches facilitate the upgrading process by generating change–rules based on various input data, such call dependency, text similarity, software metrics, *etc.* However, existing approaches do not provide 100% precision and recall because of the limited set of input data that they use to generate change–rules. For example, an approach only considering text similarity usually discovers less change–rules then that considering both text similarity and call dependency with similar precision. But adding more input data may increase the complexity of the change–rule generating algorithms and make them unpractical. We propse MOFAE (Multi-Objective Framework API Evolution) by modeling framework API evolution as multi-objective optimization problem to take more input data into account while generating change–rules and to control the algorithmic complexity.**

***Keywords*-framework evolution; API evolution; multi-object problem; search based software engineering;**

I. Introduction

Software frameworks[1] and libraries are widely used in software development for cost reduction. They evolve constantly to fix bugs and meet new requirements. In theory, the Application Programming Interface (API) of the new release of a framework should be *backward-compatible* with its previous releases, so that programs linked[2] to the framework continue to work with the new release. In practice, the API syntax and semantics change [2], [3] For example, from JHotDraw 5.2 to 5.3, method CH.ifa.draw.figures.LineConnection.end() was replaced by LineConnection.getEndConnector(); such a change may have direct consequences on a program using the JHotDraw framework, such as compile errors, or indirect ones, such as runtime errors if invoking a deleted method using reflection.

To prevent backward-compatibility problems, developers may delay or avoid using a new release. Yet, if they want to benefit from new features or security patches, they must evolve their programs. This *evolution process* often requires a lot of effort because developers must dig into the documents and–or source code of the new and previous releases to understand their differences and to make their programs compatible with the new release.

[1]Without loss of generality, we use the term "framework" to mean both frameworks and libraries.

[2]We refer readers to [1] for a discussion on the links between frameworks and programs.

Consequently, many approaches have been developed to ease this evolution process and reduce the developers' effort. Some require that the framework developers do additional work, such as providing explicit change–rules with annotations [4], or that they record API updates to the framework. [5], [6], [7]. To reduce the framework developers' involvement, others automatically identify change–rules that describe a matching between *target methods*, *i.e.* methods existing in the old release but not in the new one, and *replacement methods* in the new release by analyzing some input data of the two releases, such as call dependencies, text similarity, software metrics, *etc.* [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21].

However, framework developers may not be willing to build change–rules manually or use specific tools. Also, existing approaches are limited by the input data that they use to generate change–rules. For example, approaches using call dependency analysis cannot detect change–rules for target methods not used within the previous releases of the framework and program; others based on text similarity analysis cannot identify replacement methods if the names of the old and new releases are not similar enough. Still the hybrid approaches cannot detect the changes not related the input data that they use. Like [20] combining call dependency and text similarity analyses, it is not able to generate change–rules which could be detected by analyzing inheritance dependency because inheritance is not considered in their approach. It is also not straightforward to add a new input data to existing approaches because it may require major modification of their algorithms.

Furthermore, many of current approaches, like [6], [15], [16], [18], [20], generate change–rules only including the replacement methods, while others [9], [13], [21] introduce a set of candidates from which programmers choose the right replacement methods. The advantage of the former is that programmers can get the replacement methods directly, but if programmers have doubt about the change–rules, the latter can provide some helpful alternatives.

Therefore, we propose an approach MOFAE (Multi-Objective Framework API Evolution) to model framework API evolution as a Multi-Objective Optimization Problem (MOOP) [22]. We define each input that we will consider in the algorithm of MOFAE as an objective, then apply a Multi-Objective Optimization Algorithms (MOOA) to compute a Pareto front [23] of the possible change–rules considering

1063-6897/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPC.2011.43

all the input data to allow developers to choose preferred one from them. MOFAE will make change–rule generating algorithm more extendable to additional input data.

II. Research Hypotheses

The research hypotheses of my Ph.D. thesis are:

H1: Modeling framework API evolution as a MOOP will generate change–rules with similar or better precision and recall comparing to existing approaches based on the same input data.

H2: The size of the Pareto front of possible change–rules is small enough for human to process (*e.g.*, less then 10).

H3: Modeling framework API evolution as a MOOP will facilitate extending change–rule generating algorithm to consider more input data.

H4: Including more input data will improve the precision and recall of generated change–rules.

III. Methodology

To detect the chang-rules for framework API evolution problem, developers must consider multiple input data, such as call dependency, text similarity, inheritance relations, *etc.* Modeling the problem as a MOOP could facilitate the decision making process of developers.

Let us assume that there is a framework that we will generate the change–rules between two releases. A is the set of input data that we consider during the generation:

$$\mathbf{A} \quad = \quad \{a_1, ..., a_n\}$$

The set of target methods T is:

$$\mathbf{T} \quad = \quad \{t_1, ..., t_m\}$$

For each target method t_i, there is a set of possible replacement methods or candidate methods C_i, where x is the size of C_i:

$$C_i \quad = \quad \{c_{i1}, ..., c_{ix}\}$$

For each input a_i, we define a measure $m_i(t_j, c_{jk})$ $(1 \leq i \leq n)$ between at_j and a c_{jk}:

$$M(t_j, c_{jk}) \quad = \quad \{m_1(t_j, c_{jk}), ..., m_n(t_j, c_{jk})\}$$

For two methods c_{jx} and c_{jy}, c_{jx} dominates c_{jy} iff:

$$\begin{aligned} & m_i(t_j, c_{jx}) \geq m_i(t_j, c_{jy}) \ \forall \ i \in \{1, ..., n\} \\ \wedge \quad & \exists \ i \in \{1, ..., n\} \mid m_i(t_j, c_{jx}) > m_i(t_j, c_{jy}) \end{aligned}$$

MOFAE finds the Pareto front for each t_j in which all the c_{jk}s are not dominated by the others in C_j.

The main steps of MOFAE are:

- choose the input data that we will consider in change–rule generating as objectives to formulate framework API evolution as a MOOP. For example, if we take call dependency and text similarity into account, the problem is modeled as two-objective optimization problem.
- define the measures of the the input data. For text similarity, we could choose Levenshtein Distance [24] or Longest Common Subsequence to measure and compare it. For other input data, such as call dependency, we must define the measure and use static analysis to obtain the data to compute it.
- apply a MOOA to generate a Pareto front of change–rules. Pareto front is a way to present the solutions of MOOPs. It is a set of solutions that no other solutions outperform when considering all objectives [23]. Using the example of call dependency and text similarity, the Pareto front could contain two change–rules: one with the best call dependency measure and the other with the highest text similarity.

IV. Evaluation

To test the four research hypotheses, we will implement MOFAE in a Java System, conduct controlled experiments, evaluate and compare the results with AURA [20] and SemDiff [9]. We choose them, because, on average, AURA's result has better recall and similar precision than the other works and SemDiff presents its result in the format of candidate method set with high precision.

To compare with the two approaches, we assume that MOFAE detects the correct change–rules, if the replacement methods are included in the Pareto front, but not necessary to be the only element(s). We think this assumption holds if **H2** is true.

For **H1**, we will configure MOFAE to consider the same input data as AURA, *i.e.*, call dependency and text similarity, apply it to the same target systems, evaluate and compare the results.

For **H2**, we will count the sizes of the Pareto fronts of all target methods, compute the distribution and the average, verify if most of them are too large for human process, *e.g.*, greater than 10.

For **H3** and **H4**, we will add more input data to MOFAE, count the the effort that we spend to modify existing code and do the same experiment as we do for **H1**.

We will build a benchmark repository of the studied systems and results and share it with the research community.

V. Related work

A. Framework Evolution

Several approaches help developers evolve their programs when the frameworks that they use change. We studied them and identified eight features. Table I summarizes their differences regarding to these features.

Capturing API Updates: Existing approaches of capturing API-level changes [4], [5], [6], [7] either require the framework developers' efforts by manually specifying the change–rules or by requiring them to use a particular IDE to automatically record the refactorings performed.

Approaches	Features							
	FDI	**Result Presentations**	**Main Matching Techniques**	**One-to-many Rules**	**Many-to-one Rules**	**Simply-deleted Rules**	**Auto-matic**	**Thres-holds**
Chow *et al.* [4]	Yes	A	A	No	No	Yes	No	No
SemDiff [9]	No	Set	CD	Yes	Yes	Yes	No	Yes
Godfrey *et al.* [13]	No	Set	TS, M, and CD	Yes	Yes	Yes	No	Yes
CatchUp! [6]	Yes	Rules	N/A	No	No	No	Yes	No
M. Kim *et al.* [16]	No	Rules	TS	No	Yes	Yes	Yes	Yes
S. Kim *et al.* [15]	No	Rules	TS, M, and CD	No	No	No	Yes	Yes
Schäfer *et al.* [18]	No	Rules	CD	No	No	No	Yes	Yes
Diff-CatchUp [21]	No	Set	TS and SS	Yes	Yes	Yes	No	Yes
AURA [20]	No	Rules	CD and TS	Yes	Yes	Yes	Yes	No

Table I
FEATURE COMPARISON. (A = ANNOTATION, CD = CALL DEPENDENCY, FDI = FRAMEWORK DEVELOPER INVOLVEMENT, M = METRICS, N/A = NOT APPLICABLE, TS = TEXT SIMILARITY, SS = STRUCTURAL SIMILARITY)

Result Presentations: Automatic approaches [6], [15], [16], [18], [20] use change–rules including the replacements directly. For non-automatic approaches [9], [13], [21], they present a set of possible replacements in which programmer can choose the right(s). Chow *et al.* [4] requires framework developers to write annotations in their code to show the change–rules.

Matching Techniques: Exsiting approaches [9], [13], [15], [16], [18], [20], [21] use one or more matching techniques form call dependency, text similarity, metrics, structural similarities of logical design model.

Many-to-one , One-to-many and Simply Deleted: Semi-automatic approaches [9], [13], [21] and AURA [20] are able to report these change–rules.

Automatic and Thresholds: Automatic approaches [15], [16], [18], except CatchUp! [6] and AURA [20], use thresholds to keep a balance between precision and recall.

Framework Evolution between Different Frameworks: Nita and Notkin use Twinning [25] to adapt different Java frameworks with similar functionalities. Zhong *et al.* present MAN [26] to map APIs between Java and C#.

B. API Analysis

Exemplar developed by Grechanik *et al.* [27] analyzes API calls to improve the precision of relevant application searching. Kawrykow and Robillard's approach [28] detects the reimplementation of APIs of existing libraries in client programs.

C. MOOP in Software Engineering

There are approaches to solve Multi-Objective Optimization Problems (MOOPs) in software engineering with Search-Based Techniques [29]. Zhang *et al.* [30] use Multi-Objective Genetic Algorithm (MOGA) NSGA II [31] to solve Next Release Problem with two objectives: total cost and overall requirements importance score. Gueorguiev *et al.* [32] formulated project robustness problem as a MOOP with three objectives: (1) Completion Time. (2) Completion Time with New Tasks. (3) Completion Time with Delayed Tasks. They apply another MOGA SPEA II [23].

ACKNOWLEDGMENT

I would like to thank my supervisors. Their expertise, encouragement and support helped me a lot to carry out this work. I really enjoy working with them.

REFERENCES

[1] D. M. German and A. E. Hassan, "License integration patterns: Addressing license mismatches in component-based development," in *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 188–198.

[2] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2005, pp. 265–279.

[3] D. Dig and R. Johnson, "How do apis evolve? a story of refactoring: Research articles," *J. Softw. Maint. Evol.*, vol. 18, no. 2, pp. 83–107, 2006.

[4] K. Chow and D. Notkin, "Semi-automatic update of applications in response to library changes," in *: Proceedings of the 1996 International Conference on Software Maintenance*, ser. ICSM 1996. Washington, DC, USA: IEEE Computer Society, 1996, p. 359.

[5] 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*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 427–436.

[6] 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*. New York, NY, USA: ACM, 2005, pp. 274–283.

[7] C. Kemper and C. Overbeck, "What's new with jbuilder," in *JavaOne Sun's 2005 Worldwide Java Developer Conference*, 2005.

[8] G. Antoniol, M. D. Penta, and E. Merlo, "An automatic approach to identify class evolution discontinuities," in *IWPSE '04: Proceedings of the Principles of Software Evolution, 7th International Workshop*. IEEE Computer Society, 2004, pp. 31–40.

[9] B. Dagenais and M. P. Robillard, “Recommending adaptive changes for framework evolution,” in *ICSE ’08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 481–490.

[10] S. Demeyer, S. Ducasse, and O. Nierstrasz, “Finding refactorings via change metrics,” in *OOPSLA ’00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2000, pp. 166–177.

[11] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, “Automated detection of refactorings in evolving components,” in *ECOOP ’06: Proceedings of the 20th European Conference on Object-Oriented Programming*. Springer Berlin / Heidelberg, July 2006.

[12] B. Fluri and H. C. Gall, “Classifying change types for qualifying change couplings,” in *ICPC ’06: Proceedings of the 14th IEEE International Conference on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 35–45.

[13] M. W. Godfrey and L. Zou, “Using origin analysis to detect merging and splitting of source code entities,” *IEEE Trans. Softw. Eng.*, vol. 31, no. 2, pp. 166–181, 2005.

[14] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin, “Automated support for program refactoring using invariants,” in *Proceedings of the IEEE International Conference on Software Maintenance*, ser. ICSM 2001. Washington, DC, USA: IEEE Computer Society, 2001, p. 736.

[15] S. Kim, K. Pan, and E. J. Whitehead, Jr., “When functions change their names: Automatic detection of origin relationships,” in *WCRE ’05: Proceedings of the 12th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 143–152.

[16] 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*. Washington, DC, USA: IEEE Computer Society, Not Available 2007, pp. 333–343.

[17] G. Malpohl, J. J. Hunt, and W. E. Tichy, “Renaming detection,” in *ASE ’00: Proceedings of the 15th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 2000, p. 73.

[18] T. Schäfer, J. Jonas, and M. Mezini, “Mining framework usage changes from instantiation code,” in *ICSE ’08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, May 2008, pp. 471–480.

[19] 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*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 231–240.

[20] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, “Aura: a hybrid approach to identify framework evolution,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE ’10. New York, NY, USA: ACM, 2010, pp. 325–334.

[21] Z. Xing and E. Stroulia, “API-evolution support with diff-CatchUp,” *IEEE Trans. Softw. Eng.*, vol. 33, no. 12, pp. 818 – 836, December 2007.

[22] Y. Sawaragi, H. Nakayama, and T. Tanino, *Theory of multiobjective optimization*. Academic Press, 1985.

[23] E. Zitzler and L. Thiele, “Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach,” *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 257–271, November 1999.

[24] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions and reversals,” *Soviet Physics Doklady*, vol. 10, p. 707, 1966.

[25] M. Nita and D. Notkin, “Using twinning to adapt programs to alternative apis,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE ’10. New York, NY, USA: ACM, 2010, pp. 205–214.

[26] 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 - Volume 1*, ser. ICSE ’10. New York, NY, USA: ACM, 2010, pp. 195–204.

[27] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, “A search engine for finding highly relevant applications,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE ’10. New York, NY, USA: ACM, 2010, pp. 475–484.

[28] D. Kawrykow and M. P. Robillard, “Improving api usage through automatic detection of redundant code,” in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 111–122.

[29] M. Harman and B. F. Jones, “Search-based software engineering,” *Information and Software Technology*, vol. 43, no. 14, December 2001.

[30] Y. Zhang, M. Harman, and S. A. Mansouri, “The multi-objective next release problem,” in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, ser. GECCO 2007. New York, NY, USA: ACM, 2007, pp. 1129–1137.

[31] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: Nsga-ii,” *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 182–197, August 2002.

[32] S. Gueorguiev, M. Harman, and G. Antoniol, “Software project planning for robustness and completion time in the presence of uncertainty using multi objective search based software engineering,” in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, ser. GECCO 2009. New York, NY, USA: ACM, 2009, pp. 1673–1680.