APIEvolutionMiner: Keeping API Evolution under Control

André Hora*, Anne Etien*, Nicolas Anquetil*, Stéphane Ducasse*, Marco Tulio Valente†
* RMoD team, Inria, Lille, France
Email: *firstname.lastname*@inria.fr
† Department of Computer Science, UFMG, Belo Horizonte, Brazil
Email: mtov@dcc.ufmg.br

***Abstract*—During software evolution, source code is constantly refactored. In real-world migrations, many methods in the newer version are not present in the old version (e.g., 60% of the methods in Eclipse 2.0 were not in version 1.0). This requires changes to be consistently applied to reflect the new API and avoid further maintenance problems. In this paper, we propose a tool to extract rules by monitoring API changes applied in source code during system evolution. In this process, changes are mined at revision level in code history. Our tool focuses on mining invocation changes to keep track of how they are evolving. We also provide three case studies in order to evaluate the tool.**

I. Introduction

During software evolution, features are added, bugs are fixed, and source code is refactored to improve maintainability. In such cases, changes must be consistently applied over the code base to avoid inconsistencies. In Eclipse, 60% of the methods in version 2.0 were not in version 1.0 [1]. In the migration of the Pharo language from version 1.4 to 2.0, 27% of the methods in the newer version were not present in the older version. This implies that calls to methods must be consistently updated to reflect the new API and avoid further maintenance problems.

To solve such problems, researches proposed to extract rules from source code history due the existence of recurring refactorings in code repositories [2], [3]. While some work is dedicated to extract rules from code history by learning from bug-fixes [3] others focus on extracting rules from changes between major releases [1], [2], [4]. Rules can also be created by experts on the system under analysis [5], or targeted towards specific goals. However, such approaches are not suited to keep track of API evolution as they extract rules from bug-fix changes and major releases; or require the manual intervention of experts.

In this paper, we propose APIEvolutionMiner, a tool to extract API rules by monitoring invocation changes applied in source code during system evolution. In this process, changes are extracted at revision level in code history. Our tool focuses on mining invocation changes to keep track of how they are evolving. Moreover, we provide three case studies in order to evaluate the tool.

The paper is structured as follows. In Section II, we present our approach and tool. In Section III, we present case studies to evaluate our approach. In Section IV, we discuss related work, and we conclude the paper in Section V.

II. Approach and APIEvolutionMiner

Figure 1 shows an overview of our approach to support discovering rules, which is separated in three steps. In Step 1, we extract deltas from revisions in system history (Subsection II-A). Such step is done once for a system history under analysis. The next steps are part of the rule discovering process, which is supported by APIEvolutionMiner. Our rules are produced based on method calls that should be replaced. Thus, in Step 2, we select the changes related to such calls (Subsection II-B1). Finally, in Step 3, we mine the selected changes to discover rules (Subsection II-B2). Also, in Subsection II-C we discuss how to automatically produce rules.

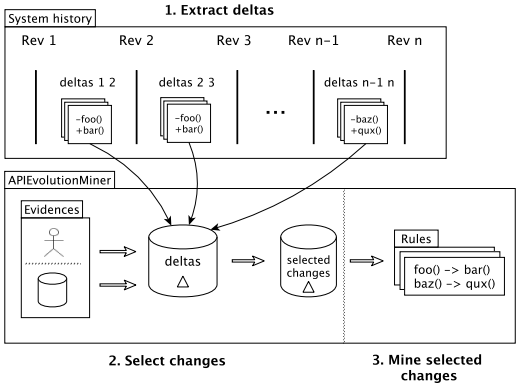

Fig. 1. Overview of our approach. Steps 2 and 3 are covered by our tool.

The tool produces rules to indicate how method calls should be replaced. There are two ways to produce rules:

1) On demand rules: Method calls that should be replaced are provided by the developer. He will receive on demand rules about how the particular method calls should be replaced.
2) Automatic rules: Method calls that should be replaced are automatically extracted from code history. The developer will receive automatically created rules about the overall API evolution. For example, rules to ensure the change patterns shown in Figure 2.

Consider two change patterns that occurred in Pharo[1] shown in Figure 2. Figure 2-I shows the adoption of a form to

[1] http://www.pharo-project.org

978-1-4799-3752-3/14/$31.00 © 2014 IEEE

retrieve a default system configuration, *i.e.,* calls to ClassOrganizer.default() are replaced by calls to Protocol.unclassified(). Figure 2-II shows the adoption of a form for testing if an object is *null*, *i.e.,* calls to equals(nil) are replaced by isNil(). We want to ensure that such changes are not lost and can be materialized as rules.

I. Replace ClassOrganizer.default() by Protocol.unclassified()

```
Diff of method mA() between rev 1 and 2:
− if (method.protocol() == ClassOrganizer.default()) {
+ if (method.protocol() == Protocol.unclassified()) {
```

II. Replace equals(nil) by isNil()

```
Diff of method mE() between rev 2 and 3:
− if (context.equals(nil)) {
+ if (context.isNil()) {
```

Fig. 2. Examples of change patterns in Pharo. '−" indicates the deleted line and "+" indicates the added line. Bold indicates the changed call. Code converted to Java-like syntax to ease understanding.

A. Extracting Deltas from Revisions

The first step of our approach is to extract deltas from the revisions, which will be used in the rule discovering process. Let a delta be a set of changes (deleted and added invocations) of a method representing the differences between two revisions. We represent a delta with predicates that describe added or deleted method invocations:

deleted-invoc(id, receiver, signature, arguments[])
added-invoc(id, receiver, signature, arguments[])

where the predicate *deleted-invoc(. . .)* represents a deleted invocation; the predicate *added-invoc(. . .)* represents an added invocation; *id* uniquely identifies the change to save its context; *receiver* is the name of the receiver[2]; *signature* is the signature of the invoked method; and *arguments* is the list of arguments, which are abstracted as "*" if they are not primitive types such as *int*, *boolean* or *null*. In addition, as a metadata, each delta has a size (the number of deleted and added invocations) and an age (the timeframe between the date it occurred and the last analyzed delta).

Notice that, as we do not rely on the type of receivers and arguments, such approach can be used in statically and dynamically typed languages. Also, it keeps the approach lightweight because there is no need to compile the analyzed source code. Figure 3 shows the deltas generated to the changes in Figure 2.

I. Replace ClassOrganizer.default() by Protocol.unclassified()

```
deleted-invoc("mA-1-2", "ClassOrganizer", "default()", [])
added-invoc("mA-1-2", "Protocol", "unclassified()", [])
```

II. Replace equals(nil) by isNil()

```
deleted-invoc("mE-2-3", "context", "equals(*)", ["nil"])
added-invoc("mE-2-3", "context", "isNil()", [])
```

Fig. 3. Deltas generated to the changes patterns in Figure 2.

B. Discovering Rules

Rules are computed from the extracted deltas, and indicate how calls should be replaced. Next, we describe the steps to discover rules, which are supported by APIEvolutionMiner.

[2]For example: implicit, local, instance variables or method names.

1) Selecting Changes: We name *evidences* the method calls that should be replaced. Let an *evidence* be a triple with the elements *receiver*, *signature*, and *arguments*:

evidence = <*receiver*, *signature*, *arguments[]*>

Given a set of related evidences, we select, for each delta: (i) the deleted invocations that match the evidences and (ii) the added invocations:

select-changes(evidences[]) ⇒
return, for each delta, the deleted invocations that match the *evidences* ∪ added invocations

For example, next, we select the changes shown Figure 3-I:

select-changes(["ClassOrganizer", "default()", []]) ⇒
deleted-invoc("mA-1-2", "ClassOrganizer", "default()", [])
added-invoc("mA-1-2", "Protocol", "unclassified()", [])

Moreover, elements of the evidence can be abstracted with "?". Thus, below, we abstract the receiver (Figure 3-II) because it is likely to be not constant:

select-changes([?, "equals(*)", ["nil"]]) ⇒
deleted-invoc("mE-2-3", ?, "equals(*)", ["nil"])
added-invoc("mE-2-3", ?, "isNil()", [])

We mine the selected changes to discover rules. Before that, we transform the selected changes in a set of transactions so that it can be used in the mining process. For example, for the previous query, we have one transaction:

select-changes([?, "equals(*)", ["nil"]]) ⇒
T_1: *"deleted ?.equals(nil)", "added ?.isNil()"*

2) Mining Selected Changes: The last step of our approach is to discover rules from the selected change transactions.

The transactions are analyzed using the data mining technique *frequent itemset mining* [6]. Given a set of transactions, this technique identifies the itemsets which are subsets of at least n transactions. It defines the *support* as the number of occurrences of an itemset. An itemset is considered frequent if its support is greater than or equal to a specified threshold called *minimum support*:

find-frequent-itemsets(transactions[], min-supp) ⇒
return the itemsets in *transactions* with *min-supp*

For example, for the transaction generated by the previous query (*i.e.,* transaction T_1), *find-frequent-itemsets([T_1], 1)* produce three frequent itemsets, each one with *support = 1*, as shown in Table I.

TABLE I
FREQUENT ITEMSETS FOR TRANSACTION T_1.

Id	Frequent itemset	Supp
I_1	*"deleted *.equals(nil)"*	1
I_2	*"added *.isNil()"*	1
I_3	*"deleted *.equals(nil)", "added *.isNil()"*	1

From the frequent itemsets, *association rules* [6] are computed. An association rule is defined as $L \rightarrow R$, where L and R are itemsets. Moreover, an association rule has a *confidence*, which is the probability of finding the R in transactions under the condition these transactions also contain L. The *confidence* is calculated as *conf($L \rightarrow R$) = supp(Itemset) / supp(L)*. Given a set of itemsets and a *minimum confidence* indicating the minimum accepted confidence, association rules are produced:

find-assoc-rules(itemsets[], min-conf) ⇒
return the association rules in *itemsets* with *min-conf*

We want to produce rules in the specific format *Evidences* → *Replacement*, where *Evidences* must include only itemsets with deleted invocations, and *Replacement* must include only itemsets with added invocations. According to the association rule definition, we can consider that *Replacement* = X − *Evidences*, thus, X must include only frequent itemsets with both deleted and added invocations. For example, in Table I, only I_3 is considered relevant to our approach, since it satisfies the condition to include both deleted and added invocations. Thus, if X is the frequent itemset I_3, *find-assoc-rules([*I_3*], 1)* produces the following rule with *confidence = 1*:

Rule: *"deleted ?.equals(nil)" → "added ?.isNil()"*

The overall process for generating rules *Evidences* → *Replacement* is sketched in Figure 4.

```
relevant-assoc-rules(evidences[], min-supp, min-conf) {
    transactions = select-changes(evidences);
    frequent-itemsets = find-frequent-itemsets(transactions, min-supp);
    relevant-itemsets = select the itemsets in frequent-itemsets that
                        includes both deleted and added invocations;
    rules = find-assoc-rules(relevant-itemsets, min-conf);
    return rules }
```

Fig. 4. Overall process to generate rules *Evidences* → *Replacement*.

C. Automatically Discovering Rules

As stated in the begin of this section, in practice, there are two ways to generate rules: on demand or automatically. We have on demand rules when the developer provides evidences to relevant-assoc-rules(...). We have automatic rules when the evidences are extracted from code history and provided to relevant-assoc-rules(...). Below we describe our heuristic to automatically discover rules.

Let *n-deleted-invocations* be the number of deleted invocations in a delta. We consider that evidences are the deleted invocations of each delta with *n-deleted-invocations* less than or equal to a specified threshold *max-deleted-invocations*. In such evidences, we abstract the receivers of non-static invocations since they are not likely to be constant as they can be, for instance, variable names. In order to have relevant evidences, they should not come from deltas with large changes since such changes are known to be related with a great amount of noise [1]. For example, from the deltas in Figure 3, two evidences are extracted:

E_1: *"ClassOrganizer", "default()", []*
E_2: *?, "equals(*)", ["nil"]*

When provided to relevant-assoc-rules(...), the evidences produce the next rules, which represent their change patterns:

R_1: *"del ClassOrganizer.default()" → "added Protocol.unclassified()"*
R_2: *"del ?.equals(nil)" → "added ?.isNil()"*

D. Tool Browser

Figure 5 shows the main browser of APIEvolutionMiner[3]. It is implemented in the Moose Platform[4]. In the Input pane the developer can set the minimum support and the evidences, *i.e.,* it calls relevant-assoc-rules(...). In addition, he can also set two strategies in order to filter the produced rules: delta size and age strategies. The *delta size* filters the deltas to be analyzed according to its size while the *age* filters according to its age (*e.g.,* deltas from the last six months).

The Association Rule pane shows the associations rules with confidence and support generated by the given input. When a rule is selected, the Delta pane displays a list with all the deltas in which the rule was found. In addition, there is an alternative displaying of the deltas using distribution map [7], [8], where the box represents a delta, the color represents the commiter of the delta, and the size represents the size of the delta. The idea is to have an overview of the distribution of the deltas with respect their commiter and size. For example, in the distribution map presented, we see that the deltas have three commiters (*i.e.,* three different colors) and distinct sizes. When a delta is selected (either clicking on the listing or on the boxes), the Diff pane shows original code from which the delta came. It shows the old revision of the code on the left side and newer revision on the right side. Other panes, such as to support discovering automatic rules, are not described in the paper due to limit of space.

III. Case Study

We selected three open-source Smalltalk systems as our case studies: Pharo, Seaside[5], and Roassal[6], from which 74, 66, and 29 rules were found, respectively. They are large and real-world systems with relevant source code history. Pharo [9] is a Smalltalk-inspired language and environment. Seaside [10] is an open-source framework for developing web applications. Roassal is a visualization engine which graphically renders objects with interaction facilities. Next, we present some extracted rules.

Pharo. Rule isNil().ifTrue(*) → ifNil(*) is a convention about calling less methods when testing *null* objects, which makes the test clearer and shorter. It occurred in 92 deltas, 8 revisions, and in timeframe of 77 days between its first and last revision. Rule UserManager.default().currentUser() → Smalltalk.tools().userManager() is about using the class Smalltalk as a central entry point to the system. Note that in this case the old API is not necessarily deprecated; Smalltalk class acts as a facade to access the system.

Seaside. Rule ifNotNil(*) → isNil().ifFalse(*) is a system specific convention about calling more methods when testing *null* objects, with a timeframe of 500 days. Rule registerAsApplication(*) → WAAdmin.registerAsApplicationAt(*,*) is about using static method calls instead of shortcut calls. This rule has a timeframe of 589 days.

Roassal. In this system, the carriage return was first represented as String.cr(), and afterwards as Character.cr(). Thus, this generated rule String.cr() → Character.cr(). Later, the carriage return was represented as ROPlatform.current().newLine(), a convention specific for this system. Therefore, this generated rule Character.cr() → ROPlatform.current().newLine().

By analyzing incremental API changes patterns, we are able to discover relevant rules which represent previously unknown API updates or usage conventions. As our approach caught them, it can be used to avoid related maintenance problems.

[3]http://rmod.lille.inria.fr/web/pier/software/APIEvolutionMiner
[4]http://www.moosetechnology.org
[5]http://www.seaside.st
[6]http://objectprofile.com/#/pages/products/roassal/overview.html

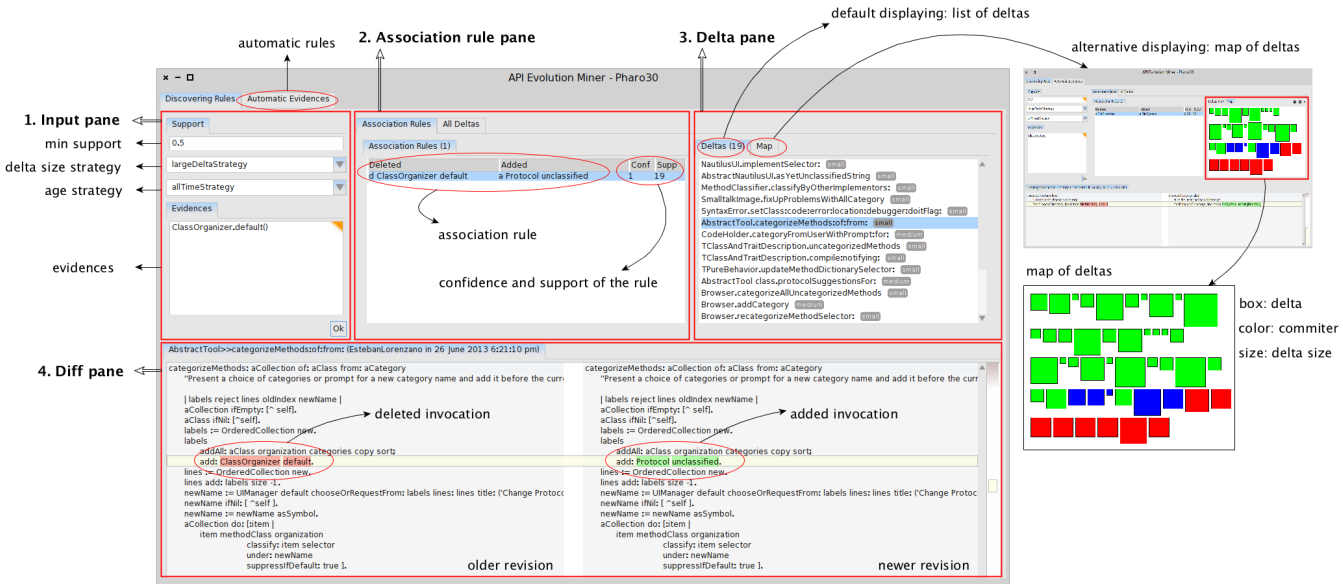

Fig. 5. APIEvolutionMiner main browser.

IV. Related Work

Traditionally, two approaches are proposed to ensure consistency of changes in source code: rules can be created by experts, or extracted from changes in code history.

Rules created by experts. A first solution is to use rules provided by static analysis tools such as PMD, Findbugs, or SmallLint. However, overall, these rules are generic, *i.e.,* not focusing on the system under analysis. Such rules can also target domains [5], or specific goals such as migration. PMD contains a set of rules (pmd.sourceforge.net/pmd-5.0.2/rules/java/migrating.html) to support migrating systems from a Java version to another by updating method calls. In FindBugs, some rules suggest the replacement of calls from an API to another, even if the replaced API is not necessarily deprecated. Overall, they focus on adjusting the use of an API to another that is better suited. However, they are generic rules, which come from previous experience of developers; getting access and capturing their knowledge into new rules is costly.

Rules extracted from code history. A possible alternative is to extract rules from code history. When analyzing source code history, information can be extracted by comparing two or more system versions [1], [2], [4], [11]. Some work mine changes only related to bug-fixes [3]. This limits the search space as it ignores ordinary commits by focusing on rules related to bug-fixes. However, relevant information might also be lost when not analyzing ordinary commits. As software evolves, naturally, not just bugs are fixed, but code is added, removed and refactored. In our previous study [12], we focused on extracting API rules based on predefined rule patterns by analyzing also ordinary commits. However, in this case, the rules are restricted to follow such patterns.

V. Conclusion

We proposed a tool to extract rules by monitoring API changes applied in source code during system evolution. Changes are mined from invocation changes at revision level to keep track of how APIs are evolving. Also, our tool can provide either on demand or automatic rules.

Acknowledgments: This research is supported by Agence Nationale de la Recherche (ANR-2010-BLAN-0219-01), FAPEMIG (process CEX-APQ-00214-11) and STIC-AmSud/CAPES (process 821).

References

[1] Y. M. Mileva, A. Wasylkowski, and A. Zeller, "Mining Evolution of Object Usage," in *European Conference on Object-Oriented Programming*, 2011.

[2] M. Kim and D. Notkin, "Discovering and Representing Systematic Code Changes," in *International Conference on Software Engineering*, 2009.

[3] S. Kim, K. Pan, and E. E. J. Whitehead, Jr., "Memories of Bug Fixes," in *International Symposium on Foundations of Software Engineering*, 2006.

[4] 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 *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

[5] A. Hora, N. Anquetil, S. Ducasse, and S. Allier, "Domain Specific Warnings: Are They Any Better?" in *International Conference on Software Maintenance*, 2012.

[6] M. Zaki and W. Meira Jr, "Fundamentals of data mining algorithms," 2012.

[7] S. Ducasse, T. Girba, and A. Kuhn, "Distribution Map," in *International Conference on Software Maintenance*, 2006.

[8] A. Hora, C. Couto, N. Anquetil, S. Ducasse, M. Bhatti, M. T. Valente, and J. Martins, "BugMaps: A Tool for the Visual Exploration and Analysis of Bugs," in *European Conference on Software Maintenance and Reengineering, Tool Track*, 2012.

[9] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker, *Pharo by Example*. Square Bracket Associates, 2009.

[10] S. Ducasse, A. Lienhard, and L. Renggli, "Seaside: A Flexible Environment for Building Dynamic Web Applications," *IEEE Software*, vol. 24, 2007.

[11] N. Meng, M. Kim, and K. S. McKinley, "Lase: locating and applying systematic edits by learning from examples," in *International Conference on Software Engineering*, 2013.

[12] A. Hora, N. Anquetil, S. Ducasse, and M. T. Valente, "Mining System Specific Rules from Change Patterns," in *Working Conference on Reverse Engineering*, 2013.

APPENDIX A
DESCRIPTION OF HOW THE DEMO WILL BE CONDUCTED

We will show how to generate rules for the case studies presented in the paper (*i.e.,* Pharo, Seaside, and Roassal) demonstrating the idea of on demand and automatic rules.

A. On demand rules

We will provide evidences to the tool in order to produce rules. We will show step by step how to produce rules and check their real occurrence in source code, *i.e.,* the changes from which the rules were learned. Thus, we present the data flow starting in the Input pane, then the Association rule and the Delta pane until the Diff pane (Figure 6). In this process, we will also present the confidence and support, and how such measures are related to the quality of the rules.

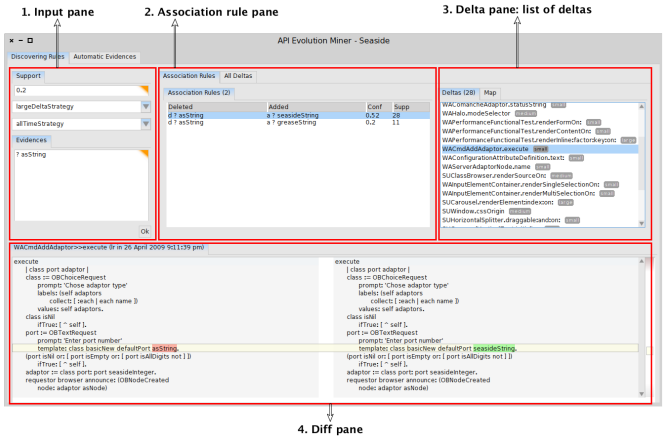

Fig. 6. Main browser.

In addition, we will show the Delta pane displaying the map of deltas (Figure 7). It provides an overview of the distribution of the deltas with respect commiters and size. For example, in the presented map, the selected rule came mostly from small and medium deltas (size of the boxes) and from four distinct commiters (color of the boxes: green, blue, red, and orange).

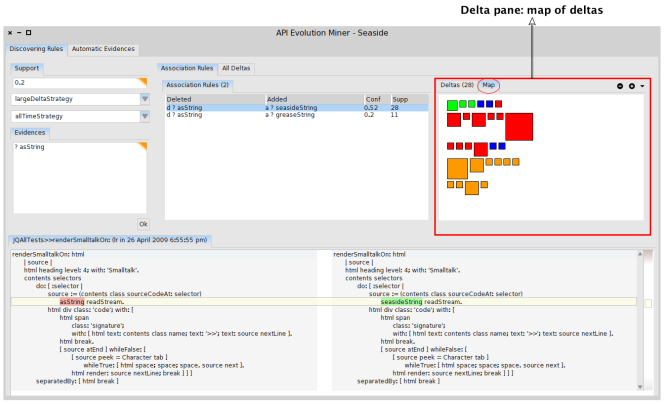

Fig. 7. Delta pane displaying the map of deltas.

B. Automatic rules

We will provide evidences to the tool in order to produce rules. However, in this case, the evidences will come from Automatic evidences pane (Figure 8), which were extracted from code history. The idea is that even if the developer is not an expert on the system under analysis, he still can generate rules in order to understand how a particular API evolved.

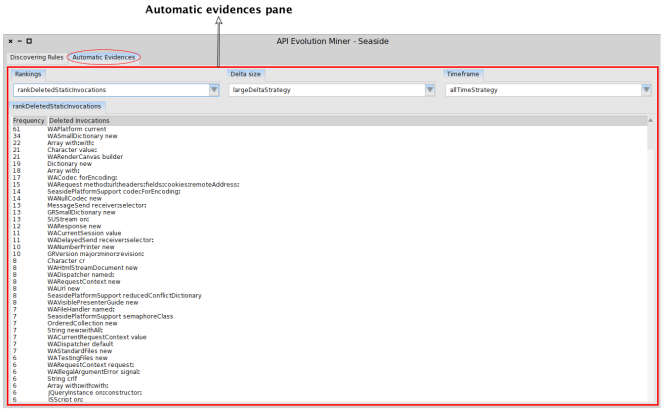

Fig. 8. Automatic evidences pane.