Automatic Detection of System-Specific Conventions Unknown to Developers

André Hora, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, Marco Túlio Valente

▶ **To cite this version:**

André Hora, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, Marco Túlio Valente. Automatic Detection of System-Specific Conventions Unknown to Developers. Journal of Systems and Software, 2015, 109, pp.192-204. 10.1016/j.jss.2015.08.007 . hal-01185837

HAL Id: hal-01185837

https://inria.hal.science/hal-01185837

Submitted on 21 Aug 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

*Manuscript
Click here to view linked References

Automatic Detection of System-Specific Conventions Unknown to Developers

André Hora[a,b,*], Nicolas Anquetil[b], Anne Etien[b], Stéphane Ducasse[b], Marco Túlio Valente[a]

[a] *Department of Computer Science, UFMG, Belo Horizonte, Brazil*
[b] *RMoD Team, Inria, Lille, France*

Abstract

In Apache Ant, a convention to improve maintenance was introduced in 2004 stating a new way to close files instead of the Java generic InputStream.close(). Yet, six years after its introduction, this convention was still not generally known to the developers. Two existing solutions could help in these cases. First, one can deprecate entities, but, in our example, one can hardly deprecate Java's method. Second, one can create a system-specific rule to be automatically enforced. In a preceding publication, we showed that system-specific rules are more likely to be noticed by developers than generic ones. However, in practice, developers rarely create specific rules. We therefore propose to free the developers from the need to create rules by automatically detecting such conventions from source code repositories. This is done by mining the change history of the system to discover similar changes being applied over several revisions. The proposed approach is applied to real-world systems, and the extracted rules are validated with the help of experts. The results show that many rules are in fact relevant for the experts.

Keywords: Automatic coding convention detection, Mining software repositories, Software evolution, Empirical software engineering

1. Introduction

During its life, a software system is subject to changes in programming conventions. These changes may have different purposes, but generally enhance code quality and ease maintenance. For example, by deciding that the constructor Double(double) in Java should be replaced by Double.valueOf(double), the system's performance is improved. The knowledge of these change conventions may take time to spread in the developer community. Developers rarely make use of deprecation annotations to explicitly mark the calls to be avoided [25], or it may be impossible to use deprecation (in

*Corresponding author
Email addresses: hora@dcc.ufmg.br (André Hora), nicolas.anquetil@inria.fr (Nicolas Anquetil), anne.etien@inria.fr (Anne Etien), stephane.ducasse@inria.fr (Stéphane Ducasse), mtov@dcc.ufmg.br (Marco Túlio Valente)

the example above the Double(double) is not deprecated), or older languages may lack the ability to deprecate anything.

Because such conventions are not linked to compilation or execution errors they can remain unnoticed. They are not used uniformly over the system and occurrences of the old form will still be found and even introduced years after the first decision was taken [8]. In this case, the benefits of the new convention, which are often to free the maintainers from unneeded cognitive burden, are lost over a large part of the system. It will increase maintenance difficulty as the unaware maintainers are left wondering why in some cases one convention is used whether in other cases it is another convention. On the other hand, we give example (in Section 5.1) where the fact that a change in convention is not uniformly applied is actually very meaningful because it points to different usage scenarii.

One proposed solution to help in this case, is to use static analysis tools, such as PMD [3], FindBugs [10], or SmallLint [26] with rules that will check and highlight the occurrences of the old convention. These tools come with a set of generic rules, such as the Double(double) example above (taken from FindBugs rules[1]), but they cannot answer to system-specific needs, which are more likely to be noticed by developers [7, 24]. In Section 2, we will give examples of conventions that are specific to a given system, for example, the Apache Ant convention stating a new way to close files. Such system-specific rules are neither linked to compilation/execution errors nor hinder correct execution of the system. This raises the problem of how these rules can be defined which may be a non trivial task. Some solutions were proposed to automatically extract system-specific rules from a system's code history, but they present limitations that make them ill-fitted to tackle this problem. For example, some of these existing solutions will rely on the hypothesis that a method has been removed and its call should be substituted by new method calls [19, 31], but this does not apply to the scenario we are considering here where the old convention does not prevent correct compilation and execution. Other solutions will rely specifically on bug fix changes to automatically correct future occurrences of these bugs or prevent their apparition [14]. Again, this does not fit our scenario where the old convention does not hinder correct execution of the system but is judged less efficient.

In this paper, we propose to automatically detect system-specific conventions unknown to some developers of the system. Our goal is to provide rules that will detect occurrences of the old convention and thus will reduce the time it takes to apply the new convention over the entire system.

Our approach is presented in Section 3. To lower the amount of noise in the candidate rules generated, we use patterns that constrain the rules to meaningful ones. We also set restriction over the occurrences of the new convention in various revisions.

We validate our approach (Section 4) on two open-source systems, Pharo[2] [1, 2] and Moose[3] [5, 23], with the help of experts of these systems. The results (presented in Section 5) show that many rules are in fact relevant and can be used to ensure faster

[1] http://goo.gl/JK3aDi
[2] http://www.pharo.org
[3] http://www.moosetechnology.org

adoption of new coding conventions.

In addition, in Section 6, we discuss specific details of implementing our approach such as whether it is better to extract rules from revisions or releases, thresholds to decide when a rule should be created, how to create new patterns, and we also compare the system-specific extracted rules with generic rules provided by a static analysis tool.

This work is an extension of our previous study [8]. First, it provides new patterns to produce rules concerning method invocation. These patterns are included in the new validation study. Second, we extended the validation by asking the opinion of an expert (previous validation was only automatic). We also provide more information on how to fine-tune the approach to a specific system with new results from our experiments with respect to the occurrence of the new convention over distinct revisions and by showing how new rule patterns might be created. Finally, we discuss the overlap between our rules and generic rules provided by static analysis tools.

Thus, the main contributions of this paper can be summarized as follows:

- We provide an extension of our previous study [8] with new patterns to extract system-specific rules.
- We provide a qualitative and quantitative (both with the help of a system expert) evaluation of the rules extracted from real-world systems.
- We provide an analysis about the overlap between our rules and rules provided by static analysis tools as well as about the creation of rules.
- We discuss rule extraction at revision and release level.
- We discuss and illustrate how new patterns can be added to create new rules.

We close this paper with a discussion of the threats to the validity of our experiments (Section 7), a presentation of related work (Section 8), and the conclusion (Section 9).

2. Motivating Examples

This section presents concrete cases in which system-specific rules extracted from source code history would be helpful to developers, and the advantages of our approach. For this purpose, we consider two examples extracted from real systems where change conventions have been performed by developers for different reasons. They illustrate two important change conventions characteristics: (i) several years later they are still occasionally applied by aware developers, and, (ii) they are very specific to each system.

First, in Apache Ant[4], a convention stating a new way to close files, *i.e.,* calls to InputStream.close() should be replaced by calls to FileUtils.close(*), was introduced in the system in 2004 to improve maintenance centralizing the knowledge on closing files. In this case, it is not possible to deprecate the old way of doing things corresponding to

[4] http://ant.apache.org

the Java InputStream.close() method. After the addition, **this specific convention continued to be occasionally applied in source code in the following six years** (*i.e.,* 37 changes in 100 possible cases) by Apache Ant developers aware of it [8]. Whereas this change convention has been introduced to improve maintenance, it in fact degrade it since the developers have to know that the two pieces of code can be used. Moreover, with time, the reasons of the change conventions and the change conventions themselves may have been forgotten. A new developer will not know which method to use, and, consequently, the invocations to the old one may increase instead of disappearing.

Second, in Roassal (a visualization library that runs in several platforms[5]), a convention was introduced to improve portability. The convention stated that calls to Collection.ifEmpty(*) should be replaced by calls to Collection.isEmpty().ifTrue(*) because Collection.ifEmpty(*) is platform-specific. **We detected such convention being occasionally applied in Roassal source code during one year and half**. Moreover, in Pharo, the language in which Roassal is written, the convention is the opposite. Thus, applying the Pharo-specific rule to Roassal, or the Roassal-specific rule to Pharo would actually decrease their code quality.

In both cases, if these conventions were better known to developers, or if a recommendation rule existed, the changes would have been applied in source code at once or in a shorter time frame and only in the adequate context, not in another system. It is laborious for developers to create rules for each change convention they performed. Our idea is therefore to use the fact that a change conventions are parsimoniously applied to automatically detect them in source code history, and describe them as rules. These two examples have some characteristics in common:

- They can be described as change rules: as they are recurrent and follow a specific format, they can be described as change rules that can be automatically applied by static analysis tools;
- They are system-specific. This can be seen as a drawback as defining such rules is typically a costly task that needs, here, to be repeated for each system. However, previous research [7, 24] showed that system-specific rules are more relevant than generic ones, more likely to be followed by developers and having better chances to remove errors in the code;
- They are spread over different revisions: the changes occur in different revisions (commits) of the systems, differently from changes related to API evolution involving, for example, class or method renaming, which cannot be spread over revisions;
- They are not "hard errors", raising issues at compilation or execution and which would get more chances to be rapidly noticed. They are rather coding convention that can ease (if followed) or hinder (if not followed) the maintenance by freeing the maintainers from unneeded cognitive burden.

[5] `http://objectprofile.com/ObjectProfile.html`

Previous researchers took advantage of the fact that similar source code changes are recurrent to support bug-discovering, or API evolution. In the bug-discovering context, researchers restrict their analysis to bug-fix changes (*e.g.,* [15, 14, 22, 28, 29]). In the API evolution context, researchers normally restrict their analysis to detect how deleted methods are replaced [31, 19] or are limited to produce *one-replaced-by-one* rules [4, 27]. In both cases, they do not focus on the detection of change conventions, and occurrences over different revisions are not considered. In fact, the examples we presented are neither related to bug-fix changes nor involved with deleted methods: they are system-specific conventions incrementally applied by developers over different revisions.

Our approach describes these changes as rules by (i) analyzing all revisions in source code history, (ii) considering, in this process, methods not removed from the system, and (iii) taking into account their occurrence in different revisions to generate better rules. Our final goal is to automatically produce system-specific rules.

3. Mining System-Specific Rules

In this section we present our approach, which extracts system-specific rules by monitoring API changes found in source code history of the system. Our approach is divided in three steps, as shown in Figure 1. First, we extract replacement facts from the source code history and store them in a database (Subsection 3.1). A replacement is composed of an addition and a removal of some code. Here we will exemplify it with method invocation replacements. Second, we look for some pre-defined patterns in the database of replacements and we generate candidate rules (Subsection 3.2). Finally, we discard some candidate rules that have a high probability of being false positives (Subsection 3.3). Each of these three steps of the approach is fully automated and does not require the presence of experts.

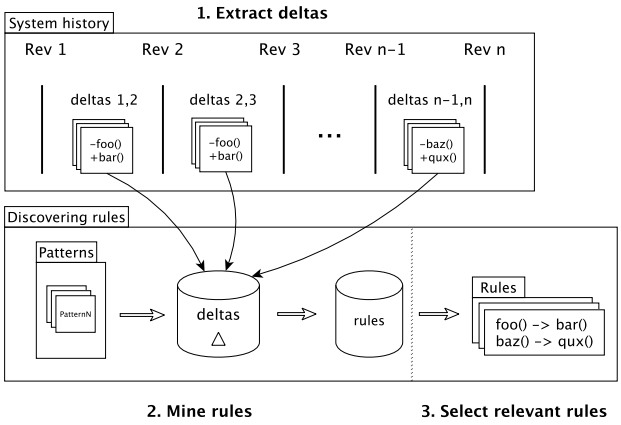

Figure 1: Overview of our approach.

Before detailing our approach, we present an overview about it. Consider the examples shown in Figures 2 and 3 that occurred in the Pharo programming language. Figure 2 shows a replacement of the static method call RPackageOrganizer.default() by RPackage.organizer(). Figure 3 shows a replacement to improve code legibility, *i.e.,* calls to Collection.at(3) are replaced by calls to Collection.third().

Diff between revisions 1 and 2 of method foo()
− rpackage = **RPackageOrganizer.default()**;
+ rpackage = **RPackage.organizer()**;

Figure 2: Replacement example of RPackageOrganizer.default() by RPackage.organizer() in Pharo.

Diff between revisions 3 and 4 of method bar()
− if (coll.**at(3)**) { . . .
+ if (coll.**third()**) { . . .

Figure 3: Replacement example of at(3) by third() in Pharo.

Notice that the replaced methods might still exist in the system. In Figure 2, the old method call, RPackageOrganizer.default() should only be used in specific cases and by some classes.[6] In Figure 3, the new method call, someCollection.third(), is considered a rule to better convey code intention. These changes occurred several times in different revisions, and they testify the efforts to use a better API.

This also means that these conventions are different from refactoring operations such as renaming a method, moving it elsewhere, or changing its signature. In our approach there is no difference between renaming a method or "simply deciding" that another one should be invoked instead of it, as is the case when one recommends to use Java Double.valueOf(Double) rather than the constructor Double(Double). In the first case, the renamed method "disappears" from the system, in the second case, the constructor is still available but the recommendation is to not use it. For us, both cases will result in removal of the old invocations and addition of the new ones, *i.e.,* a code replacement.

3.1. Extracting Deltas (Step 1)

The first step of our approach is to extract changes from the revisions. When comparing the differences between changed source code, one needs to define what should be analyzed in the code. For example, one can keep track of fine-grained changes (*i.e.,* adding or removing conditional statements, modifying expressions), or more coarse-grained changes (*i.e.,* adding or removing classes, methods, method invocations). While the first ones play an important role in the context of bugs discovering [14, 22, 28, 29], they have a small role in discovering systematic changes, for which the second ones are better suited since they do not take into account low level changes [12, 15].

[6] As stated in `http://goo.gl/ch8Ba2`.

In this work, our approach extracts rules, for example, from method call changes (*e.g.,* invocation to method foo() is replaced by a call to method bar()). In order to do so, we need to extract the correct data from the source code history of a system.

We iterate over all the revisions of the system under analysis, starting in the second revision. In each iteration, we detect pair of methods that are present in both current and previous revision (*i.e.,* methods that did not change their signature). For each pair of method, we consider a *delta* to be the set of deleted and added calls between the method pair. We represent a delta with predicates that describe each deleted or added method call:

*delta := [predicate]**
predicate := deleted-call(args) or added-call(args)
args := [id, receiver, signature, static]

where, the predicate *deleted-call(. . .)* represents a deleted invocation; the predicate *added-call(. . .)* represents an added invocation; *id* uniquely identifies a change context (*i.e.,* the changed full method name and the revision); *receiver* is the receiver of the invocation; *signature* is the signature of the invoked method (for the arguments, the value is represented if they are primitive types such as int, boolean or null, otherwise the type is represented; this is done to obtain more precise rules); and *static*[7] is a keyword *isStatic* or *notStatic* that states if the invocation is static. There is a balance that we need to strike into account when fixing the delta size: larger deltas will extract more rules but with less precision while smaller deltas will extract less rules but with more precision. Small deltas between revisions are preferable to avoid the noise that can be found in large ones [15, 20]. Thus, to avoid the problem of large diff size between methods, making it difficult to extract relevant information, we always select deltas involving less than five deleted or added calls.

Discovering other type of rules may imply extracting other data (*e.g.,* inheritance), which is not in the scope of this work. Notice that the approach is independent from the programming language because the facts (here invocations) are represented in a language independent model (the added-call and deleted-call predicates). In Figures 4 and 5, we present the deltas generated by the changes in Figures 2 and 3.

Deltas between revisions 1 and 2 of method foo()
deleted-call(“foo()-rev2”, “RPackageOrganizer”, “default()”, “isStatic”)
added-call(“foo()-rev2”, “RPackage”, “organizer()”, “isStatic")

Figure 4: Deltas generated for the changes in Figure 2.

3.2. Mining Rules (Step 2)

The second step of our approach is to mine rules from the extracted deltas. To add new relevant rules, we define patterns that the rules must follow. These patterns will

[7] In our previous study [8] we did not use the “static” variable because for Java we could resolve the type of the receive and for Pharo we only took into account changes related to static calls (then, again, we have the type of the receiver).

Deltas between revisions 3 and 4 of method bar()
deleted-call(“bar()-rev4”, “coll”, “at(3)”, “notStatic”) added-call(“bar()-rev4”, “coll”, “third()”, “notStatic”)

Figure 5: Deltas generated for the changes in Figure 3.

limit the search space, and, thus, the extraction of noisy rules. In particular, such patterns were inspired by existing change rules found in static analysis tools. For example, Table 1 presents some real-world rules and patterns that inspired us in this study.

Table 1: Example of change rules found in FindBugs and SmallLint static analysis tools.

Tool	Rule Description	Rule	Pattern
FindBugs	Use the nextInt of Random rather than nextDouble to generate a random integer	Random.nextDouble() → Random.nextInt()	Change invocation, same receiver, static
SmallLint	Consider using isNil when testing null objects to improve legibility	obj.equals(nil) → obj.isNil()	Change invocation, same receiver, non-static

Our solution automatically creates new rules related to method call replacement. A classical change rule in SmallLint states that invocations to Object.equals(nil) should be replaced by invocations to Object.isNil(). Assume that this rule has been applied in source code; so for each replacement, our approach generates a delta similar to:[8]

deleted-call(“mtd()-revX”, “obj”, “equals(nil)”, “notStatic”)
added-call(“mtd()-revX’, “obj”, “isNil()”, “notStatic”)

The deltas are used as a database in which we want to find instances of predefined patterns. For example, the previous change follows the pattern where the receiver (and *id*) remains the same (*i.e.,*obj) while the method call changes (*i.e.,* from equals(nil) to isNil()). We could query our database to search instances of such pattern. For example, a SQL-like query would be:

select deleted-call.signature, added-call.signature
from deleted-call, added-call
where deleted-call.id = added-call.id **and** deleted-call.receiver = added-call.receiver
and deleted-call.static = “notStatic” **and** added-call.static = “notStatic”

As shorthand notation, we use the following pattern to represent the previous query:

examplePattern(deletedSignature, addedSignature) =
deleted-call(id, receiver, deletedSignature, “notStatic”) and
added-call(id, receiver, addedSignature, “notStatic”)

[8]Each with their respective *id*.

The variables within the predicates are used to ensure the constraints of the pattern. In this paper, we use the same variable in both deleted and added predicates to ensure the same id, receiver or signature (*e.g.,* in the example above *id* and *receiver* are the same in both predicates). We use different variables in both predicates to ensure different id, receiver or signature (*e.g., deletedSignature* and *addedSignature*, such that *deletedSignature* $\neq$ *addedSignature*). The predicates *deleted-call(...)* and *added-call(...)* are used with the logical operator *and*, thus two or more predicates can be grouped to form a pattern. The part before the equals represents the name and the output of the pattern, which is formed by a subset of variables used in the predicates. For example, the example above is named *examplePattern* and it outputs rules in the format: deletedSignature $\rightarrow$ addedSignature, where the left hand side (LHS) presents what should be deleted and the right hand side (RHS) presents what should be added. The output for the previous delta is equals(nil) $\rightarrow$ isNil().

The following list of patterns has been defined in order to detect system-specific rules. This list is not exhaustive and a discussion about additional patterns is proposed in Subsection 6.3. Next, we present each pattern.

Pattern 1: Change receiver and invocation, static
pattern1(deletedReceiver, deletedSignature, addedReceiver, addedSignature) =
deleted-call(id, deletedReceiver, deletedSignature, "isStatic") *and*
added-call(id, addedReceiver, addedSignature, "isStatic")
Example
FileDirectory.default() $\rightarrow$ FileSystem.workingDirectory()

Pattern 2: Change invocation, same receiver, static
pattern2(receiver, deletedSignature, addedSignature) =
deleted-call(id, receiver, deletedSignature, "isStatic") *and*
added-call(id, receiver, addedSignature, "isStatic")
Example
SystemNavigation.default() $\rightarrow$ SystemNavigation.new()

Pattern 3: Change receiver, same invocation, static
pattern3(deletedReceiver, addedReceiver, signature) =
deleted-call(id, deletedReceiver, signature, "isStatic") *and*
added-call(id, addedReceiver, signature, "isStatic")
Example
SystemChangeNotifier.uniqueInstance() $\rightarrow$ SystemAnnouncer.uniqueInstance()

Pattern 4: Change invocation, same receiver, non-static
pattern4(deletedSignature, addedSignature) =
deleted-call(id, receiver, deletedSignature, "notStatic") *and*
added-call(id, receiver, addedSignature, "notStatic")
Example
obj.noMoreNotificationsFor() $\rightarrow$ obj.unsubscribe()

Pattern 5: Change double invocation, same receiver, non-static
pattern5(deletedSignature1, deletedSignature2, addedSignature1, addedSignature2)=
deleted-call(id, receiver, deletedSignature1, "notStatic") *and*
deleted-call(id, deletedSignature1, deletedSignature2, "notStatic") *and*
added-call(id, receiver, addedSignature1, "notStatic") *and*
added-call(id, addedSignature1, addedSignature2, "notStatic")
Example
obj.vm().getSystemAttribute(1001) → obj.platform().name()

Patterns 1, 2 and 3 represent the cases where a static invocation is replaced by another. Patterns 4 and 5 cover the replacement of non-static invocations. These patterns were kept because they produced little noisy rules. In Subsection 6.3 we discuss the creation of additional patterns.

We recall that the patterns should be applied in small deltas in order to avoid noisy rules. In this work, we apply Patterns 1 to 4 in deltas with up to three deleted or added calls, and Pattern 5 in deltas with four deleted or added calls. This is done to avoid false positives in many-to-many replacements since Pattern 4 can be considered a sub-pattern of Pattern 5.

From the delta shown in Figure 4, we find a rule based on Pattern 1: RPackageOrganizer.default() → RPackage.organizer(), and from the delta in Figure 5, we find a rule based on Pattern 4: at(3) → third().

3.3. Selecting Relevant Rules (Step 3)

In the source code history of real systems many rules may be found. Our goal is to provide rules which are likely to be system-specific and relevant. The usage of patterns ensures a certain quality since they enforce a structure on the rules, *i.e.,* an old and a new call. However, even with such precautions and even if the added rules are clearly system-specific, they may not all be relevant. The relevance of the rules can be stated by experts. Nevertheless, the idea of the approach is to automatically detect relevant rules corresponding to change conventions without using experts but only relying on code history analysis. If it is difficult to state that a rule is relevant, it is safer to say that it is not or that currently we have not enough information to decide. Indeed, changes that occur in only one revision are likely to simply capture method or class renaming done, for example, with the support of refactoring tools provided by current IDEs or to capture a localized change or refactoring. These changes are not in the scope of our work. In contrast, we tackle change modifications for which there exist no other tool to automatically and largely apply them.

Thus, we consider rules as relevant if they occur in two or more revisions. We decided to use at least two revisions to limit false positives (noisy rules). For this purpose, we compute the number of revisions in which the changes expressed in the rule occurred and keep the rules corresponding to changes occurring in two or more revision. Obviously, the more revisions there are in which the changes occur, the more relevant the corresponding rule is. Additional discussion on this point is given in Subsection 6.2.

4. Validation Experiment

In this section we detail our research questions and the experiments that test them. We first present the proposed main research questions (Subsection 4.1). Then, we present the context of our experiments detailing the case studies and the evaluated change rules (Subsections 4.2 and 4.3). Finally, in Subsections 4.4 and 4.5, we formalize the experiments design used to answer the main research questions. Complementary research questions will be treated in Section 6.

4.1. Research Questions

We propose research questions to assess the rules generated by the proposed approach. We assess the change rules correctness and whether they are likely to find violations in source code that developers would fix.

Assessing Rules Correctness. We evaluate whether the rules are correct according to the opinion of an expert (it has to be noticed that the expert presence is required only for the scientific evaluation but not during the concrete use). Thus we propose a first research question:

RQ1 *Are the rules correct to the system expert?*

Assessing Rule Violations. The rules may be classified as correct but produce no violation when applied to source code. To complement RQ1, we also evaluate whether violations identified by the change rules are likely to be fixed, as suggested by the rule. This will also be validated with the help of an expert. For example, a violation for the rule at(3) $\rightarrow$ third() is a piece of source code that still contains a call at(3) instead of the call third(); if the expert decides that this piece of code should be fixed, he will replace by third(). Thus, we propose another research question:

RQ2 *Are the violations "real" ones (*i.e., *that the experts want to fix)?*

4.2. Case Studies

The *context* of the experiment is real systems for which source code history is available. We need real systems to ensure that our experiment is meaningful, and we need source code history to extract our change rules. Moreover, it is fundamental to have access to the experts of the systems to receive relevant assessment.

In this work, we selected two open-source systems, Pharo [1, 2] and Moose [5, 23] to perform our empirical studies. They have the advantage of being large, real-world and non-trivial systems, with a consolidated number of developers as well as relevant source code history. Also, they have different missions working in different domains.

Pharo is an open-source Smalltalk-inspired dynamically typed language and environment. It can be compared to the Java SDK and includes the implementation of all features inherent to an object-oriented language (collections, exceptions, primitive types, etc.) as well as an IDE and several tools. It is currently used in many industrial and research projects.[9] The latest analyzed version has 374 KLOC, 3,246 classes, and is supported by 37 developers.

[9] `http://consortium.pharo.org`

Moose is an open-source platform for software and data analysis written in Pharo. It is composed of several tools to deal with meta-modeling; frameworks to build visualizations, diagrams, interactive browsers; it also includes tools to support common software maintenance tasks such as code duplication detection, identifying dependency cycles, among others. It is currently supported by several research groups around the world[10], and also adopted in industrial projects. The latest analyzed version has 210 KLOC, 2,617 classes, and is supported by 45 developers.

We extracted rules from source code changes that occurred during the evolution of Pharo 1.4 to 2.0 (435 revisions from April 2012 to March 2013). Then, we applied such rules in the last release of Pharo 2.0 itself and in Moose. Although we claim that the rules are system-specific, when they are about the Pharo public API, they are also relevant to Pharo clients. Moose is a Pharo client and should thus benefit also from Pharo's new conventions.

We have access to an expert of the Pharo environment, which is fundamental to receive relevant assessment about the rules and their violations and to validate the approach. The expert selected to validate the rules and the violations is a core developer and release master of the system. He has worked on it since the first version and for 10 years on a preceding system (Squeak) from which Pharo is derived. Results were evaluated violation by violation, sometimes with the help of documentation.

Additionally, in a previous study [8] we experimented with three Java systems: Apache Ant, Tomcat and Lucene. Ant is a tool for automating software build processes, Tomcat is a web server and servlet container, and Lucene is an information retrieval software library. This analysis was intended to compare the specific rules created by our approach with generic ones found in static analysis tools. We evaluated whether our approach can be used to improve the set of generic rules provided by these tools, and then provide better rules to developers. This experiment is discussed in Section 5.

4.3. Detecting Rules

We obtain the rules by mining Pharo code changes which incrementally occurred in revisions between versions 1.4 and 2.0. As small deltas between revisions are preferable to avoid the noise that can be found in large ones [15, 20], we select deltas involved in less than five deleted or added invocations. There is a total of 6,513 deltas; from such deltas, 4,272 (65,6%) have less than five deleted or added invocations. Therefore, they represent a relevant amount of the deltas.

In this process, changes are represented as the deltas described in Subsection 3.1 and stored in a database. From this database of deltas, we generated the rules as described in Subsection 3.2.

As shown in Table 2, this process generated a total of 426 rules considering all patterns. Because this was too much for the expert evaluation, we ranked the rules generated by each pattern by the number of distinct revision they appear in. Then, we only analyzed the top-15 (*i.e.,* the first 15 in the ranking) rules generated by each pattern; this was done to reduce the amount of rules to be manually analyzed by an expert. To detect the rules relevant for our study (*cf.* Subsection 3.3), we selected

[10] `http://www.moosetechnology.org/docs/publications`

from the top-15 the ones that occurred in two or more revisions. In the case that some rule(s) in the top-15 and the 16th rule occurred in the same amount of revisions, they were not considered; this was done to ensure a maximum of 15 rules per pattern and then facilitate the validation by the expert. This process generated at the end 45 rules considering all patterns.

Table 2: Rules obtained from Pharo.

Pattern	1	2	3	4	5	Total
All Rules	25	31	49	304	17	426
Rules ≥ 2 revisions	14	11	3	13	4	45

4.4. Experiment for RQ1: Assessing Rules Correctness

RQ1 *Are the rules correct to the system expert?*

With the help of an expert we validate the rules according to their correctness. We asked the expert to classify the rules as correct or incorrect, a *correct* rule being one that he believes would describe a valid modification to apply in the source code.

4.5. Experiment for RQ2: Assessing Rule Violations

We detail this experiment with the methodology proposed by Wohlin et al [30].

4.5.1. Hypotheses Formulation

RQ2 *Are the violations "real" ones (*i.e., *that the experts want to fix)?*

H_0^2 *Number of violations before and after fixing are the same.*

H_a^2 *Number of violations before and after fixing are not the same.*

4.5.2. Variable and Subject Selection

The *subjects* for this experiment are the *violations* generated by rules. First, we take the last version of Pharo and Moose, and we compute the number of violations generated by each rule; this will generate a sample, namely *violations before fixing*. From such sample, we remove the violations that, according to the expert, should be fixed exactly as suggested by the rule; this will generate another sample, namely *violations after fixing*.

The *independent variable* is the rule. It is categorical and takes two values: before or after fixing the violations. The *dependent variable* (measured) is the number of violations for each rule.

4.5.3. Experiment Design

We want to compare the two generated samples (*violations before fixing* and *violations after fixing*). Thus, we use a paired setting, which means the rules composing one sample (before fixing the violations) are the same than those composing the other sample (after fixing the violations). We use the Wilcoxon test which is used for assessing whether one of two samples tends to have smaller/larger values than the other. It can be used when the participants are the same in each sample. The null hypothesis is that the median of violations is the same for both samples. The tests will be performed at the 5% significance level (*i.e.,* $\alpha = 0.05$).

We also report the *effect size* which measures the distance between the null hypothesis and alternative hypothesis, and is independent of sample size. Effect size value 0.1, 0.3 and 0.5 are considered small, medium and large effects, respectively.

Notice that for this experiment we do not present recall. That would imply knowing all the conventions of the systems under analysis, which is not feasible in practice due to the size of the systems. The decision of not computing recall is also shared by related studies such as [4, 14, 27].

5. Experiment Results

In this section, we present the results of our empirical study and discuss them. All the results (the generated rules and the evaluation by the expert) are available for download.[11]

The whole process to produce rules took no longer than one hour, for each system. Notice that, execution time was rather long in this case because it was the first analysis of the systems. It meant we had to download the full code history, compute the deltas, and process them. For day to day use, one can compute the deltas and process them incrementally (see the discussion in Section 6.2). In our approach, all the steps can be done off-line, by nightly jobs. In that case, execution time is negligible.

5.1. Evaluating RQ1: Assessing Rules Correctness

RQ1 *Are the rules correct to the system expert?*

Table 3 shows that 62% (28 out of 45) of the analyzed rules were correct according to the expert. In Pattern 1, 79% (11 out of 14) of analyzed rules were correct while in Pattern 3, 1 out of 3 were correct. Some patterns are clearly better than others, *e.g.,* Pattern 1 and 5 have 77% correctness. The incorrect rules were mostly noisy rules, which are likely to occur when they are extracted from deltas not related to change conventions. In such deltas, method calls not involved with change conventions tend to get intermingled with real rules [15]. The outcome of this experiment is that a relevant amount of rules are correct according to the expert. Next, we present examples of correct and incorrect rules.

[11] `https://goo.gl/PyvSMF`

Table 3: RQ1: Assessing rules correctness.

Pattern	1	2	3	4	5	Total
Relevant rules	14	11	3	13	4	45
Correct rules	11 (79%)	6 (55%)	1 (33%)	7 (54%)	3 (75%)	28 (62%)

Table 4 presents some correct rules generated for Pharo. For instance, in the rule RPackageOrganizer.default() → RPackage.organizer(), the method RPackageOrganizer.default() should only be used in specific cases as stated in a comment into the method definition of RPackageOrganizer.default(). The rule vm(). getSystemAttribute(1001) → platform().name() clearly improves legibility.

Table 4: Examples of Pharo rules.

Pattern 1	FileDirectory.default() → FileSystem.workingDirectory() RPackageOrganizer.default() → RPackage.organizer()
Pattern 2	ZnCharacterEncoder.forEncoding(*) → ZnCharacterEncoder.newForEncoding(*) SystemAnnouncer.current() → SystemAnnouncer.uniqueInstance()
Pattern 3	DataStream.initialize() → MCDataStream.initialize() SystemChangeNotifier.uniqueInstance() → SystemAnnouncer.uniqueInstance()
Pattern 4	getSource() → sourceCode() noMoreNotificationsFor(*) → unsubscribe(*)
Pattern 5	vm().getSystemAttribute(1001) → platform().name() vm().getSystemAttribute(1003) → platform().subtype()

Some of the generated rules were also incorrect (or not blindly applicable) for different reasons. The rule FileSystem.workingDirectory() → FileDirectory.default() was generated when the correct one is in fact its opposite (also generated, see 1st rule in Table 4). The incorrect rule was generated due to unrelated rollbacks applied on the source code. In this case, the correct rule was much more frequent, and, thus, easy to be detected.

Other rules were not incorrect but should not be applied blindly as there are cases where the old form is also valid. For example, the rule MCHttpRepository.locationUserPassword(*,*,*) → MCHttpRepository.location(*) should not be applied when the HTTP access does require *user* and *password* information. Similarly, the rule ComposableModel.new() → DummyComposableModel.new() came out from a specific convention to ease testing of ComposableModel.

We believe the existence of such rules, that should not be applied blindly, actually reinforce the need for applying coding convention as widely as possible. In the case of these rules, not applying the change is meaningful and points to different use scenarios. In this case, the maintainers get useful information from the fact that there are two different invocation conventions (notice that as aforementioned the correct rule is more frequent, and, thus, easy to be detected). Such cases should not be mistaken with those where the changes were not applied because the developers were not aware of the new

convention.

All the correct rules were implemented in the static analysis tool SmallLint (the static analysis tool for Smalltalk), and they are publicly available[12] to support developers. Furthermore, the discovering of 28 new system-specific rules represents a significant addition to the set of rules provided by SmallLint, which originally includes only 19 generic change rules that were created by experts.

5.2. Evaluating RQ2: Assessing Rule Violations

RQ2 *Are the violations "real" ones (*i.e., *that the experts want to fix)?*

H_0^2 *Number of violations before and after fixing are the same.*

H_a^2 *Number of violations before and after fixing are not the same.*

We extracted two samples of violations of the rules for Pharo and Moose, the first sample *before fixing* and the second sample *after fixing* the violations. Table 5 shows the number of rules that produced at least one violation and the total number of violations in each sample. In the last analyzed Pharo release, 8 rules generated at least one violation, producing a total of 21 violations (*before fixing*). The expert pointed that, from such violations, 10 should be fixed, so only 11 violations remained (*after fixing*). In the last analyzed Moose release, 7 rules generated at least one violation, producing a total of 37 violations (*before fixing*). The expert pointed that all violations should be fixed, thus, 0 violations remained (*after fixing*). Applying the Wilcoxon test in the samples gives a *p-value* < 0.01, then we reject the null hypothesis and consider that the number of violations before and after fixing are not the same. Moreover, the *effect size* is $= 0.56$, which indicates a large effect.

Table 5: RQ2: Assessing violations before and after fixing the real ones. Rules refers to the number of rules that produced at least one violation.

System	Rules	Violations	
		Before fixing	After fixing
Pharo	8	21	11
Moose	7	37	0
Total	15	58	11

We conclude that the rules are pointing to violations in source code. In total, 15 rules generated violations, producing a total of 58 violations from which 47 (81%) were real ones.

Fixing a violation generated by a rule is usually a fast and easy task if the developers knows the system. The rules consist in a LHS which is the current code and a RHS which is the suggested replacement. This can be done automatically if the user validates the application of the rule.

[12] www.smalltalkhub.com, Project: FindBugs, Package: MiningLintRules-PharoMigration

5.3. Java Case Studies

In our previous work [8], we evaluated the proposed approach on the Java systems Ant, Tomcat and Lucene. The goal of this study was to compare our system specific rules with generic ones provided by static analysis tools. In order to complement the results of this paper, we present the main research questions of our previous study and discuss them.

RQ3 *Are specific warnings more likely to point to real violations than generic warnings?*

The outcome of this experiment is that specific warnings are in fact more likely to point to real violations. This was true for all the case studies. As expected, in general, tools to detect coding standard violations produce too many false positives [11, 13, 24]. In this case, precision of generic warnings remained between 0.015 and 0.07 (which is coherent with previously published results [7, 13, 14]), while precision of specific warnings remained between 0.12 and 0.49.

However, rules are not equal in identifying real violations, *i.e.,* some rules performed better than others. Thus, we also studied whether specific rules are more likely to point to real violations than generic ones.

RQ4 *Are specific rules more likely to point to real violations than generic rules?*

This was true for Tomcat. We could not show that the specific rules performed better than the generic ones for Ant and Lucene. This was mostly because warnings generated by some generic rules in these systems were almost all fixed during the experiment timeframe.

An example of rule generated for Apache Ant was presented in the motivation section: the convention to close files, where calls to InputStream.close() should be replaced by calls to FileUtils.close(InputStream). As mentioned, this convention was introduced in the system in 2004, but in practice it has never been fully adopted as, even six years later (2010), the refactoring was still being applied. Our approach caught it and, thus, can be used to avoid similar maintenance problems.

In Tomcat, we detected rules defined by FindBugs such as "DM_NUMBER_CTOR: Method invokes inefficient Number constructor; use static valueOf instead"[13]. This rule is intended to solve performance issues and it states that using valueOf is approximately 3.5 times faster than using constructor. In fact, we detected such rules because Tomcat developers have been using FindBugs over time. Even if there was an effort to fix such violations, they were not completely removed. This means that developers may not be aware of common refactorings even when static analysis tools are adopted. Also, the great amount of violations generated by such tools in real-world systems is not easy to manage [24, 11, 13]. Our approach also caught this refactoring and again it can be used to avoid similar problems where changes are not consistently applied.

In Lucene, system specific rules were related, for example, to structural changes (*e.g.,* replace Document.get() by StoredDocument.get()) and to internal guidance to have

[13] goo.gl/JK3aDi

better performance (*e.g.,* replace Analyzer.tokenStream() by Analyzer.reusableTokenStream(), replace Random.nextInt() by SmartRandom.nextInt()). Overall, the analysis also produced rules related to Java API changes such as the replacement of calls to the classes Vector to calls to ArrayList, Hashtable to Map, and StringBuffer to StringBuilder, which were incrementally fixed by developers, and, thus, also detected by our approach.

6. Complementary Research Questions

In this section we present complementary research questions to our study. First, we compare rule extraction at revision and release level. Second, we discuss the creation of rules with respect to their frequency over different revisions. Third, we discuss the creation of additional patterns in order to complement the proposed ones. Finally, we discuss the overlap of our rules with predefined generic change rules.

6.1. Comparing Rule Extraction at Revision and Release Level

In the proposed approach we extract rules from changes at revision level. The motivation to do that came from real world examples such as the ones presented in Section 2, in which recurrent changes were spread over different revisions of the systems as well as from related studies [4, 19]. However, another solution to extract rules is at release level, *i.e.,* considering changes between releases rather than between revisions. Thus, in this subsection we compare rule extraction at revision and release level.

CRQ1 *Is it better to extract rules considering revision or release level?*

In order to produce rules at release level, we considered the changes between Pharo releases 1.4 and 2.0, *i.e.,* discarding the revisions between them. Notice that when release level is adopted, the fine-grained changes (*i.e.,* commits) existing between the two analyzed releases may be lost. As a result, when comparing two major releases, more noise can be found.

Table 6 shows the number of correct rules extracted at revision and release level. At revision level, 28 correct rules were extracted (as already presented in RQ1). At release level, only 12 correct rules were extracted; for pattern 5 none of the rules obtained at revision level were produced at release level. The smaller amount of rules at release level occurs due to the larger size of changes between the releases. In such larger changes, method calls not involved with change conventions tend to get intermingled with real rules [15]. **This confirms that, in our case, extraction at revision level gives better results than at release level**.

Table 6: Number of correct rules extracted at revision and release level.

Pattern	1	2	3	4	5	Total
Revision level	11	6	1	7	3	28
Release level	1	5	1	5	0	12

6.2. Assessing When Rules Should be Created

In our experiments we extracted rules from a specific time frame, *i.e.,* for revisions between Pharo 1.4 and 2.0. Then, we validated such rules in the latest Pharo and Moose releases. However, in practice, there should be no clear time frame about what should be analyzed in the past code history. Therefore, to support these cases, in this subsection we discuss the creation of rules considering when a change is frequent enough to be considered as a rule.

CRQ2 *When is a change frequent enough to be considered a rule?*

This frequency can impact the quality of the produced rules as well as the amount of generated rules. Thus, depending on the goal of the developer (*e.g.,* to produce rules with better precision or to produce more rules), different frequencies can be adopted. Next, we study the impact of such frequency to obtain rules.

6.2.1. Process to learn and evaluate rules

To automatically assess when rules should be created, we need to learn the rules and evaluate them incrementally revision by revision. To learn the rules from source code history, we use the approach developed by Kim et al [14] on navigating through revisions to extract information. It suits well since it works by learning from changes in revisions. The idea is that we walk through the revision history of a project *learning* rules and *evaluating* at each revision how well our approach works when using only the information available for that revision. We learn a rule when it occurs in f different revisions. We evaluate at revision n the rules learned from revisions 1 to $n-1$. If a fix in revision n matches the learned rule, *i.e.,* revision $n-1$ has a call to the LHS of the rule and revision n replaces it by a call to RHS, we have a true positive (TP) violation. If a fix in revision n matches the LHS, but not the RHS, we have a false positive (FP) violation. We can measure the precision of a rule from the portion of violations predicted correctly over all violations, *i.e., precision = TP/(TP + FP).*

6.2.2. Discussion

Table 7 shows the precision and the number of rules obtained over the frequency f such that $2 \leq f \leq 9$ for our case study. For example, if we say that rules are created when the same change occurs over two different revisions (*i.e.,* $f = 2$), then 104 rules are generated and they have a precision of 25%. As expected, the greater the frequency, the more precise are the generated rules, but the smaller the number of generated rules.

From Table 7 we can also compute deltas for the precision and the number of rules. For example, moving from $f = 2$ to $f = 3$ improves the precision by 36% but reduces the number of generated rules by 58% (which is the greatest loss in number of rules). Moving from $f = 3$ to $f = 4$ improves the precision by 85% (which is the greatest gain in precision) but reduces the number of generated rules by 46%. Also, moving from $f = 6$ to $f = 7$ changes neither the precision nor the number of generated rules.

We observe that the precision tend to be greater and the number of generated rules tend to be smaller. **If the number of generated rules is an important goal, then we should not choose a large** f.

Table 7: Evaluation of the frequency (f) to create rules.

Frequency (f)	2	3	4	5	6	7	8	9
Precision	25%	34%	63%	71%	72%	72%	75%	75%
Rules	104	43	23	12	8	8	6	6

In Table 8 we compare the precisions of the rules generated for each frequency. This comparison is done by applying the Mann-Whitney test on each pair of rule samples. For example, the *p-value* 0.01 (first line and last column) means that the precision of the rules obtained with $f = 9$ is statistically greater at the 1% level than the precision of the rules obtained with $f = 2$. Also, the *p-value* 0.43 (last line and last column) means that we cannot decide whether the precision for $f = 9$ is greater than the precision for $f = 8$. We marked with * results that are significant at the 10% level and with ** results that are significant at the 5% level.

By analyzing the first line in the table, we see that the precision of rules obtained with $f = 2$ is statistically smaller than any other sample (they are all marked). When analyzing the second line, we see that the precision of rules obtained with $f = 3$ is statistically smaller than the ones obtained with $f = 8$ and $f = 9$. The same occurs in the third line: the precision of rules obtained with $f = 4$ is statistically smaller than the ones obtained with $f = 8$ and $f = 9$. However, when analyzing the fourth line (*i.e.,* $f = 5$ in **bold**), we see that no other sample is statistically greater than the one obtained with $f = 5$. The same occurs for the subsequent lines. Therefore, it means that the precision of the rules obtained from $f = 5$ to 9 are equivalent. In such samples, the precision remains between 71% and 75% (as we see in Table 7). It means that, choosing between $f = 5$ to 9 would bring equivalent results. **If precision is an important goal, we should choose a large f, but only up to a certain point (f=5 in our experiments).**

Table 8: *p-values* comparing the precisions of the created rules. ** *p-value* < 0.05, * *p-value* < 0.10.

Frequency (f)	3	4	5	6	7	8	9
2	0.07*	0.05*	0.07*	0.02**	0.02**	0.02**	0.01**
3	-	0.36	0.23	0.11	0.11	0.07*	0.05*
4	-	-	0.25	0.11	0.14	0.05*	0.04**
5	-	-	-	**0.36**	**0.43**	**0.22**	**0.17**
6	-	-	-	-	0.60	0.39	0.43
7	-	-	-	-	-	0.37	0.27
8	-	-	-	-	-	-	0.43

6.3. Evaluating Additional Patterns

In Subsection 3.2 we proposed five patterns in order to detect system-specific rules. In practice, other patterns than the proposed ones can be created. Therefore, we ask:

CRQ3 *What other patterns can be created considering single method invocation and how relevant are them?*

For this experiment and to limit the scope, we will consider all possible patterns with one single method invocation. Table 9 shows a complete list of possible patterns for single method invocation. Recall that a pattern must include one added and one removed predicate. We use a shorthand notation as compared to the one presented in Subsection 3.2 to facilitate the comparison between the patterns.

Table 9: Exhaustive list of patterns for single method invocation. The patterns annotated with * are the ones adopted in this study. *rules* means the rules that occurred in two or more revisions. *cor* means correct rules. *s* means static and *ns* means non-static.

	deleted-call(...)			added-call(...)			rules	cor.	% of cor.
1*	deletedRec	deletedSig	s	addedRec	addedSig	s	14	11	79%
2*	receiver	deletedSig	s	receiver	addedSig	s	11	6	55%
3*	deletedRec	signature	s	addedRec	signature	s	3	1	33%
4	deletedRec	deletedSig	ns	addedRec	addedSig	ns	15	0	0%
5*	receiver	deletedSig	ns	receiver	addedSig	ns	13	7	54%
6	deletedRec	signature	ns	addedRec	signature	ns	-	-	-
7	deletedRec	deletedSig	s	addedRec	addedSig	ns	10	2	20%
8	receiver	deletedSig	s	receiver	addedSig	ns	-	-	-
9	deletedRec	signature	s	addedRec	signature	ns	5	0	0%
10	deletedRec	deletedSig	ns	addedRec	addedSig	s	3	0	0%
11	receiver	deletedSig	ns	receiver	addedSig	s	-	-	-
12	deletedRec	signature	ns	addedRec	signature	s	15	0	0%

Each group of pattern has three patterns in order to check all the possible variations of receivers and signatures. The first in each group (*i.e.,* Patterns 1, 4, 7 and 10) is about changing receiver (from *deletedRec* to *addedRec*) and invocation (from *deletedSig* to *addedSig*); the second (*i.e.,* Patterns 2, 5, 8 and 11) is about changing invocation (from *deletedSig* to *addedSig*) and keeping the same receiver; and the third in each group (*i.e.,* Patterns 3, 6, 9 and 12) is about changing receiver (from *deletedRec* to *addedRec*) and keeping the same invocation.

The first group of patterns (*i.e.,* Patterns 1 to 3) is about the replacement of static methods, the second group (*i.e.,* Patterns 4 to 6) covers the replacement of non-static methods, the third group of patterns (*i.e.,* Patterns 7 to 9) describes the replacement of static by non-static methods, and the fourth group (*i.e.,* Patterns 10 to 12) is about the replacement of non-static by static methods.

The twelve patterns are presented in Table 9 to facilitate the comprehension on how they were produced. However, Patterns 6, 8 and 11 do not make sense in our context. Pattern 6 is about changing receiver while keeping the same invocation for non-static methods, which cannot produce relevant rules since the non-static invocations are the same (*e.g.,* it would produce rules such as foo() $\rightarrow$ foo()). Patterns 8 and 11 are about changing invocation while keeping the same receiver while swapping between static and non-static methods, which cannot occur since the same receiver cannot be static and non-static.

The last three columns of Table 9 present the number of rules that occurred in two or more revisions, the correct ones and the percentage of correct rules, respectively. The patterns annotated with * (*i.e.,* Patterns 1, 2, 3, 5) are the ones with best performance. They are already the patterns described in Subsection 3.2 and used in our experiment in Section 5 (note that we also used another pattern with more than one invocation, so it does not appear here).

The patterns not annotated with * have lowest performance; they produced too many incorrect rules. The same explanation of RQ1 can be used in the cases where the rules were incorrect: they came out from specific refactorings in source code therefore we cannot generalize them to be applied in all the system. For example, Pattern 7 produced the rule TestResult.error() $\rightarrow$ defaultTestError(), which even if it makes sense, is only applicable in the context of Test classes. These patterns did not show promising in our case study because too many rules should be analyzed in order to detect correct ones (*e.g.,* Pattern 7 in Table 9) or because they did not produced correct rules at all.

Overall, the patterns involving only the replacement of static methods (first group) performed better than the others. In fact, these patterns include the type of the receiver which gives naturally more robust rules. The patterns involving only non-static methods (second group) also performed well. In this case, it is preferable patterns related to less noisy changes such as Pattern 5 in Table 9, in which the invocation change but the receiver is the same.

6.4. Overlap with Generic Change Rules

As stated in the introduction, the static analysis tool SmallLint also provides generic change rules that were manually created by experts. It contains 19 generic change rules that details how source code should be updated to improve code maintainability. Based on that we ask:

CRQ4 *Is there any overlap between the system-specific rules we generated and the generic ones provided by SmallLint?*

We found that there is no overlap between our rules and the generic change rules provided by SmallLint. This is similar to the results provided by Kim et al [14] in which the intersection of bug-detection rules generated by their approach and rules found in the Java static analysis tool PMD were exclusive. One reason for this result is that the generic change rules were not applied in the analyzed case study over different revisions or they were applied in large commits, so our approach was not able to detect them.

7. Threats to Validity

7.1. Construct Validity

The construct validity is related to whether the measurement in the study reflects real-world situations. In our study, the main threat is the validation of the rules.

As an error in this process would bias the results, our rules were manually validated with the help of an expert (which is hard to have for real-world systems) to decrease

the possibility of bias. In fact, many previous studies (*e.g.,* [27, 19, 31]) do *not* adopt the validation with experts, *i.e.,* the authors of the studies validate the rules themselves, which may introduce bias.

Another threat is that we are acquainted with the system expert. However, we reinforce that the expert has worked on Pharo since the first version and for 10 years on the preceding Pharo system. Based on that, we believe that we could not find a better expert for such analysis. For example, not all the rules were correct for him; in fact, he only marked as correct the rules that made sense for him as core Pharo developer.

Moreover, we address that Moose was initially written by one of the authors of the paper in its conception. This threat is alleviated because since then, around 10 years passed and Moose has evolved and became a broadly used tool with several developers around the world. From the Moose repository control website, we can see that it has now 45 contributors.[14] Therefore, we believe that the fact that one of the authors has initially contributed to Moose does not influence our results.

7.2. Internal Validity

The internal validity is related to uncontrolled aspects that may affect the experimental results. In our study, the main threat is the possible errors in the implementation of our approach causing the generation of wrong rules.

Apart from the validation by the expert presented in this paper, the rules generated by our approach have been (i) used by several members of our laboratory in different systems and (ii) divulged in the Moose open-source software reengineering mailing list such that developers of this community can use it, thus, we believe that the risks of this threat are reduced.

Moreover, the patterns used to extract rules from source code history may be an underestimation of the real changes occurring in commits: some changes are more complex, only introducing new code or only removing old code. We do not extract rules from such cases, and they might also represent relevant source of information. However, the patterns do reflect generic change rules found in static analysis tools.

7.3. External Validity

The external validity is related to the possibility to generalize our results. In our study, the main threat is the representativeness of our case studies.

Pharo and Moose are credible case studies as they are open-source, real-world and non-trivial systems with a consolidated number of developers and users. They also come from different domains and include a large number of revisions. Despite this observation, our findings – as usual in empirical software engineering – cannot be directly generalized to other systems, specifically to systems implemented in other languages or to systems from different domains. Closed-source systems, due to differences in the internal processes, might also have different properties in their commits. Finally, small systems or systems in initial stage may not produce data sufficient to generate rules.

[14] `http://smalltalkhub.com/#!/~Moose`

8. Related Work

System-Specific Rules

This work is an extension of our previous study [8]. In that study, we focused on extracting rules for the Patterns 1, 2 and 3 and comparing the mined rules with predefined rules provided by PMD and SmallLint. We extracted and validated the rules for the Java systems Ant, Tomcat and Lucene, and for Pharo. As a result, the mined rules were shown to be statistically more precise than PMD and SmallLint rules. Overall, such work detected several relevant rules as we discussed in Subsection 5.3. The current work differs from the previous one in several points. We have now five patterns while the previous study had three. The research questions are different: while the previous study compared the change rules with rules provided by static analysis tools, in the current work there is no comparison with static analysis tools in the research questions, instead, we focus on the correctness of rules and whether they are likely to point to real violations. In the current work we have the support of the expert to validate the change rules while in the previous the rules were validated automatically. Furthermore, in the current study we discuss the overlap between our change rules and predefined generic rules and the creation of rules with respect to its frequency over different revisions.

API Migration

Some work has been proposed to support during API evolution and migration [9, 6, 32]. Nguyen *et al.* [21] propose LibSync that uses graph-based techniques to help developers migrate from one framework version to another. In this process, the tool takes as input the client system, a set of systems already migrated to the new framework as well as the old and new version of the framework in focus. Using the learned adaptation patterns, the tool recommends locations and update operations for adapting due to API evolution.

Dagenais and Robillard [4] present a tool (SemDiff) that suggests replacements for framework elements accessed by client systems based on how a framework adapts to its own changes. Schäfer et al [27] propose to mine framework usage change rules from client systems. They produce rules by comparing two versions of a class using the framework. These studies are intended to produce *one-replaced-by-one* rules; this is done in order filter out false positives. In contrast, our study is not restricted to produce *one-replaced-by-one* rules (see for example Pattern 5, which produces *two-replaced-by-two* rules); we filter out false positive rules using patterns and considering their spread over different revisions. Moreover, the goal of our study is very different: we detect rules about conventions, and *not* API migration. Our approach can also be seen as a generalization of such related studies, in which more flexible rules can be generated.

Wu et al [31] propose an approach (AURA) that combines call dependency and text similarity analyses to produce evolution rules. They extract rules by comparing two major versions of the framework. Meng et al [19] propose a history-based matching approach (HiMa) approach to support framework evolution. The rules are extracted from the revisions in code history together with comments recorded in the evolution history of the framework. These studies focus on supporting framework clients during a migration while we focus on new rules about conventions. Moreover, such studies

use addition and deletion of methods (instead of method calls) as the basis for detecting rules. That is to say that they verify how methods deleted in one revision or release were replaced in the next one. As a result, they cannot produce rules about conventions (*i.e.,* when both methods are available to be used) as proposed by our approach.

Bug Discovering

Kim et al [14] aim to discover system-specific bugs based on source code history. They analyze bug-fixes changes extracting involving numeric and string literals, variables and method calls, which are then stored in a database. Our approach and the related work are intended to find system-specific changes but in very different contexts. There are several differences between our study and the related one. First, the input is different. While the authors check revisions related to bug-fixes in the learning process, we check *all* the revisions. As a result, they focus on rules in the context of bug-fixes while we focus on conventions spread over revisions. Second, the way rules are presented to developers is distinct. Our approach produces rules while the related work stores bug-fix changes in a database. As a result, their work is not intended to give rules beforehand for developers (they provide "on-demand rules"), in contrast with ours, which is intended to give rules for developers. Finally, the goal of both studies is very different. As stated before, the related work is intended to support developers to discover system-specific bugs while ours is intended to detect system-specific conventions unknown to the developers. In summary, to filter out noise in the data mining process (which produce many false positives), they concentrate on bug-fixes. This means they discard a lot of information that we are using. If the related study worked with as many data as us, they would produce many false positives because they would not have filtering like our approach.

Williams and Hollingsworth [29] focus on discovering warnings which are likely real bugs by mining code history. They investigate a specific type of warning (checking if return value is tested before being used), which is more likely to happen in C programs. They improve bug-finding techniques by ranking warnings based on historical information. While the authors investigate a single known type of warning, we are intended to discover new system-specific types of violations. Another similar research in the sense that authors use historical information to improve ranking mechanism is proposed by Kim and Ernst [13]. They propose to rank warnings reported by static analysis tools based on the number of times such warnings were fixed over the history. Again, they focus on defined rules while we focus on new system-specific rules that are not included in static analysis tools.

Livshits and Zimmermann [15] propose to discover system-specific usage patterns over code history. These patterns are then dynamically tested. to support discovering such pattern, they use the data mining technique Apriori. Results show that the usage patterns, such as method pairs (*e.g.,* lock() must happen with unlock()), can be found. While the related work extracts usage patterns to understand how methods should be invoked, we extract invocation changes patterns to understand how invocations should be updated.

Other studies focus on extracting information by analyzing the differences between major versions. Nguyen et al [22] aim to discover recurring bug-fixes by analyzing system two versions. They propose to recommend the fix according to learned bug-fixes.

Meng et al [18] study systematic edits in source code related to bug-fixes and feature addition. They aim to find relevant location and make correct edits in source code in order to support developers. Our work is not restricted to bug-fix analysis and we also extract rules at revision level. Mileva et al [20] focus on discovering changes that must be systematically applied in source code. These changes are obtained by comparing two versions of the same system, determining object usage, and deriving patterns. They compare object usage models and temporal properties from each version. By learning systematic changes, they are able to find places in source code where changes were not correctly applied. We focus on rule related to API changes to improve static analysis while the related work focuses on object usage. Sun et al [28] propose to extend a commercial static analysis tool by discovering specific defects. They focus on mining a graph, with data or control dependencies, to discover specific defects. While they extract data from a single version, we extract from code history. We focus on mining invocation changes to produce change rules.

Some studies investigate rule recovering from execution traces [16, 17]. While they extract rules via dynamic analysis of a single system version to produce temporal rules (*e.g.,* every call to *m1()* must be preceded by a call to *m2()*), we extract rules via static analysis of changes from incremental versions to produce change rules (*e.g.,* every call to *m1()* must be replaced by a call to *m2()*).

9. Conclusion

In this paper, we proposed to automatically extract system-specific conventions unknown to the developers. In this process, we extract information from incremental revisions in source code history and the rules are based on predefined patterns that ensure their quality.

We validated our approach on open-source systems with the help of an expert. For such validation, we had the support of a Pharo expert which was very valuable to provide assessment about the change rules. In our previous study, we validated our approach in Java systems showing that relevant rules were also extracted. In addition, in this paper, we compared the extracted rules with predefined generic change rules provided by a static analysis tool and we discussed the creation of rules with respect to their frequency over different revisions. We reiterate here the most interesting conclusions we derived from our study:

1. A relevant amount of the change rules (62%, 28 out of 45) were correct to the expert in our Pharo case study. The discovering of 28 new system-specific rules represents a significant addition to the set of rules provided by the static analysis tool SmallLint since it contains only 19 generic change rules.
2. Rules pointed to real violations in source code. In total, 15 rules generated violations, producing a total of 58 violations from which 47 (81%) were real ones.
3. Different frequencies (f) can be adopted depending on the goal of the developer to assess when rules should be created. If precision is an important goal, we should choose a large f but only up to a certain point (f=5 in our experiments).
4. There was no overlap between our rules and the generic change rules provided by SmallLint.

As future work, we plan to extract rules from other structural changes such as class access and inheritance. In addition, we plan to investigate frequent large changes to extract rules. In this case, we plan to use a data mining approach to induce the rules from the changes.

Acknowledgements

This research has been supported by grants from ANR (ANR-2010-BLAN-0219-01) and CNPq.

[1] Bergel A, Cassou D, Ducasse S, Laval J (2013) Deep into Pharo. Square Bracket Associates

[2] Black A, Ducasse S, Nierstrasz O, Pollet D, Cassou D, Denker M (2009) Pharo by Example. Square Bracket Associates

[3] Copeland T (2005) PMD Applied. Centennial Books

[4] Dagenais B, Robillard MP (2008) Recommending adaptive changes for framework evolution. In: International Conference on Software engineering, pp 481–490

[5] Ducasse S, Anquetil N, Bhatti MU, Hora A, Laval J, Girba T (2011) MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. Tech. rep.

[6] Hora A (2015) Discover and track APIs. `http://apiwave.com`, accessed: 2015-06-03

[7] Hora A, Anquetil N, Ducasse S, Allier S (2012) Domain Specific Warnings: Are They Any Better? In: International Conference on Software Maintenance

[8] Hora A, Anquetil N, Ducasse S, Valente MT (2013) Mining System Specific Rules from Change Patterns. In: Working Conference on Reverse Engineering

[9] Hora A, Etien A, Anquetil N, Ducasse S, Valente MT (2014) APIEvolutionMiner: Keeping API Evolution under Control. In: Software Evolution Week (European Conference on Software Maintenance and Working Conference on Reverse Engineering)

[10] Hovemeyer D, Pugh W (2004) Finding Bugs is Easy. In: Object Oriented Programming Systems Languages and Applications, pp 132–136

[11] Joao Araujo Filho SS, Valente MT (2011) Study on the Relevance of the Warnings Reported by Java Bug-Finding Tools. Software, IET 5(4):366–374

[12] Kim M, Notkin D (2009) Discovering and Representing Systematic Code Changes. In: International Conference on Software Engineering, pp 309–319

[13] Kim S, Ernst MD (2007) Which Warnings Should I Fix First? In: European Software Engineering Conference and Symposium on the foundations of Software Engineering, pp 45–54

[14] Kim S, Pan K, Whitehead EEJ Jr (2006) Memories of Bug Fixes. In: International Symposium on Foundations of Software Engineering, pp 35–45

[15] Livshits B, Zimmermann T (2005) DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In: European Software Engineering Conference and Symposium on the foundations of Software Engineering, pp 296–305

[16] Lo D, Khoo SC, Liu C (2008) Mining Temporal Rules for Software Maintenance. Journal of Software Maintenance and Evolution: Research and Practice 20(4):227–247

[17] Lo D, Ramalingam G, Ranganath VP, Vaswani K (2012) Mining Quantified Temporal Rules: Formalism, Algorithms, and Evaluation. Science of Computer Programming 77(6):743–759

[18] Meng N, Kim M, McKinley KS (2013) Lase: Locating and applying systematic edits by learning from examples. In: International Conference on Software Engineering, pp 502–511

[19] Meng S, Wang X, Zhang L, Mei H (2012) A history-based matching approach to identification of framework evolution. In: International Conference on Software Engineering, pp 353–363

[20] Mileva YM, Wasylkowski A, Zeller A (2011) Mining Evolution of Object Usage. In: European Conference on Object-Oriented Programming, pp 105–129

[21] Nguyen HA, Nguyen TT, Wilson G Jr, Nguyen AT, Kim M, Nguyen TN (2010) A graph-based approach to api usage adaptation. In: International Conference on Object Oriented Programming Systems Languages and Applications, pp 302–321

[22] Nguyen TT, Nguyen HA, Pham NH, Al-Kofahi J, Nguyen TN (2010) Recurring Bug Fixes in Object-Oriented Programs. In: International Conference on Software Engineering, pp 315–324

[23] Nierstrasz O, Ducasse S, Gĭrba T (2005) The story of moose: an agile reengineering environment. In: European Software Engineering Conference and International Symposium on Foundations of Software Engineering, pp 1–10

[24] Renggli L, Ducasse S, Gîrba T, Nierstrasz O (2010) Domain-Specific Program Checking. In: Objects, Models, Components, Patterns, pp 213–232

[25] Robbes R, Lungu M, Röthlisberger D (2012) How do developers react to API deprecation? The case of a smalltalk ecosystem. In: International Symposium on the Foundations of Software Engineering, ACM, pp 56:1–56:11

[26] Roberts D, Brant J, Johnson R (1997) A Refactoring Tool for Smalltalk. Theory and Practice of Object Systems 3:253–263

[27] Schäfer T, Jonas J, Mezini M (2008) Mining framework usage changes from instantiation code. In: International Conference on Software Engineering, pp 471–480

[28] Sun B, Shu G, Podgurski A, Robinson B (2012) Extending static analysis by mining project-specific rules. In: International Conference on Software Engineering, pp 1054–1063

[29] Williams CC, Hollingsworth JK (2005) Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques. Transactions on Software Engineering 31:466–480

[30] Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2000) Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers

[31] Wu W, Gueheneuc YG, Antoniol G, Kim M (2010) Aura: a hybrid approach to identify framework evolution. In: International Conference on Software Engineering, pp 325–334

[32] Xing Z, Stroulia E (2007) Api-evolution support with diff-catchup. Transactions on Software Engineering 33(12)

***Biography**

André Hora

is a Post-doctoral fellow in Software Engineering in the ASERG Group at the Department of Computer Science, Federal University of Minas Gerais, Brazil. He received a PhD degree from University of Lille-1/Inria, France, in 2014. His main research interests include software evolution, reverse engineering, software analysis, and empirical software engineering.

Nicolas Anquetil

is an assistant professor at the University of Lille-1, France, since September 2009. He obtained his PhD in 1996 from the University of Montreal, Canada. He also worked at University of Ottawa, Canada; Federal University of Rio de Janeiro, Brazil; Catholic University of Brasilia, Brazil; and, Ecole des Mines de Nantes, France. His research focuses on software evolution and maintenance at large which already included work on software re-architecturing, knowledge management for software maintenance, or software maintenance management. He is best known for his work on software re-architecturing.

Anne Etien

is been an assistant professor at Ecole Polytechnique Universitaire de Lille. She received a PhD degree from Université of Paris 1, France, in 2006. She is a member of Inria Lille and recently joined the RMoD team. Her research interests concern reengineering of complex legacy systems.

Stéphane Ducasse

is research director at Inria Lille leading the RMoD team since September 2007. During 10 years, he co-directed with Oscar Nierstrasz the Software Composition Group. He is the president of ESUG. He co-founded Synectique, a company that offers specific tools for Software analysis. He is one of the leader of Pharo: a new exciting dynamic language.

Marco Tulio Valente

received his PhD degree in Computer Science from the Federal University of Minas Gerais, Brazil (2002), where he is an associate professor in the Computer Science Department, since 2010. His research interests include software architecture and modularity, software maintenance and evolution, and software quality analysis. He is a “Researcher I-D” of the Brazilian National Research Council (CNPq). He also holds a “Researcher from Minas Gerais State” scholarship, from FAPEMIG. Valente has co-authored more than 60 refereed papers in international conferences and journals. Currently, he heads the Applied Software Engineering Research Group (ASERG), at DCC/UFMG.