Towards Pratical Reuse of Custom Static Analysis Rules for Defect Localization

Diogo S. Mendonça
Federal Center for Technological Education of Rio de Janeiro (CEFET/RJ)
diogo.mendonca@cefet-rj.br

Marcos Kalinowski
Pontifical Catholic University of Rio de Janeiro (PUC-Rio)
kalinowski@inf.puc-rio.br

ABSTRACT

[Context] Several static analysis tools allow the development of custom rules for locating application-specific defects. Although this feature is powerful and commonly available, it is not well explored in practice. Custom static analysis rules can check design and policies that are shared between applications, allowing the reuse of rules. However, the benefits, scope, and concerns that software engineers should have on reusing custom static analysis rules are unknown. [Goal] In this preliminary study, we investigate the reuse of custom static analysis rules produced by applying Pattern-Driven Maintenance (PDM). PDM is a method to locate defect patterns in web applications that produces custom static analysis rules as output. [Method] We selected a set of rules produced by a previous usage of the PDM method and applied them to other three applications in two contexts, within the same company where the rules were produced, and in other companies. [Results] We successfully reused some rules in both scenarios with minor adjustments, finding new defects to be fixed. The reuse of rules could discard from 58-90% of source code locations found by a naive search for the defects, reducing verification effort. However, the reused rules need adjustments to improve precision for defect localization, as precision ranged from 40-75%. Finally, we identified factors that have an impact on reusing custom rules. [Conclusions] We put forward that reusing customized static analysis rules can be beneficial, in particular when similarities in the architecture and programming style are observed. However, adjustment of the rules might be needed to enable effective reuse. We shared our insights and methodology on how to reuse custom static analysis rules properly.

CCS CONCEPTS

• Insert CCS text here • Insert CCS text here

KEYWORDS

Custom static analysis rules, reuse, pattern-driven maintenance

ACM Reference format:

Diogo Mendonça and Marcos Kalinowski. 2020. Toward Practical Reuse of custom static analysis rules for defect localization. In *Proceedings of Simpósio Brasileiro de Qualidade de Software (SBQS'20). ACM, São Luís, MA, Brazil, 10 pages.*

1 Introduction

Static analysis is commonly used for locating defects [1], [2]. General-purpose static analyzers that search for defects in source code are called Linters [3]. Although Linters can find a myriad of defects, they check only defects associated with the inadequate use of a programming language or the use of error-prone constructions. They do not find application-specific defects by default.

Several static analysis tools [4]–[6] allow the development of custom rules, enabling practitioners to search for application-specific defects statically. In a survey with developers, one third of them agreed that useful rules are related to a specific context, such as a particular project [7]. Although custom rules are useful and commonly available, they are not well explored in practice [8], [9]. Less than 5% of the static analysis rules used in open source projects are custom rules [8]. Furthermore, only 8% of developers reported using custom rules in practice [9].

One way to improve the usage of custom static analysis rules is by reusing them. Custom static analysis rules can check design decisions and policies, such as exception handling policies and architectural decisions, which could be shared between applications [10], [11]. In this way, custom rules can be reused, thus allowing to scale up their benefits. However, there is limited research on reusing custom static analysis rules [10], [11]. Hence, the benefits, scope, and concerns that software engineers should have on reusing custom static analysis rules are still unknown.

In this preliminary study, we investigate the reuse of custom static analysis rules created in the context of Pattern-Driven Maintenance (PDM), a method to produce rules that precisely locate defect patterns in web applications. We selected a set of rules previously produced by applying PDM in practice to an industrial web application [12] and investigate their reuse in other three web applications. We used a practical approach for reuse, i.e., by adjusting rules when necessary within practical scenarios with limited effort and complexity. We consider two contexts: outside of the scope of the company where they were produced (cross-company), and within the same company where they were produced (within-company). During this investigation, we aim at answering the following three research questions:

RQ1. (sensitivity) In which scope can custom rules be reused?

RQ2. (effect) What are the benefits and effort of reusing custom rules?

RQ3. (sensitivity) Which factors have an impact on reusing custom rules?

We successfully reused some rules in both scenarios with minor adjustments (described in detail throughout the paper), finding new defects to be fixed. The reuse of rules could discard from 58-90% of source code locations found by a naive search for the defects, reducing verification costs. However, the precision of rules for defect localization still needs improvements, having ranged from 40-75%. Finally, we observed that architecture and programming style played an essential role when reusing static analysis rules. Therefore, we recommend reusing customized static analysis rules, as long as architecture and programming style similarities are observed. Nevertheless, we emphasize that the precision and recall of the rules should be evaluated so that they can be improved for the new specific usage context. We shared our insights and methodology on how to reuse custom static analysis rules properly.

The remainder of this paper is organized as follows. Section 2 presents PDM and the rules created in its previous application as a background. Section 3 shows the methodology used in the reuse investigation. Section 4 presents the results found for our cross-company and within-company investigation scenarios. Section 5 discusses the research questions based on the results. Section 6 describes the threats to validity and how they were mitigated. Section 7 presents related work. Finally, concluding remarks are provided in Section 8.

2 Background

In this section, we explain the custom static analysis rules involved in our work and Pattern-Driven Maintenance (PDM), which was the method used to produce them. The description of PDM was first published by Mendonça *et al.* [12]. A summary description is included in this section to provide the background for understanding our investigation.

2.1 PDM Method

PDM is a method designed to produce custom static analysis rules that precisely locate defect patterns in existent web applications. As PDM was designed to be applicable to legacy systems, it does not require software documentation as input. Instead, PDM receives web server logs and application source code as inputs. Furthermore, PDM can be adapted to receive any other data source of failures that identify a specific line of code where the failure occurred, such as an issue report.

Figure 1 shows the activities of PDM collapsed into steps along with the control flow of the method. Two primary paths can be observed: the maintenance path (steps 1, 2, and 3) and the defect pattern improvement cycle (steps 4, 5, and 2). Execution of the maintenance path occurs when the web server error logs, or other sources of failures, contain new records. The selected source of failures must be monitored periodically to identify those new records.

The maintenance path includes activities to process failures data and identify defect patterns (step 1), developing custom static analysis rules to detect the latent defects (step 2), i.e., defects that were not exercised yet, not having produced a failure, and testing the identified instances and correcting the defects (step 3).

The defect pattern improvement cycle is performed when the evaluation of the rules (step 4) (e.g., based on precision and recall) does not reach acceptable levels to alert during development. The recommended level of precision of rules is at least 80%, for being well accepted by developers [9] and stable to be used [13]. The ideal level of recall is 100%, i.e., no defect is missed by the rule (no false negatives) since false negative affects the confidence that the tool can find defects [13]. PDM prioritizes recall over precision by design, starting with 100% of recall for a rule and move forward, improving precision in each improvement cycle. When precision or recall levels are not acceptable, the source code of false positives (or false negatives) is analyzed to improve the static analysis rule (step 5). Finally, there are two exit steps in the exit path of the method – rule deployment for defect alerting (step 6) and rule contingency (step 7) –, which includes using the rules and patterns only in a limited way. Futher details of the seven depicted steps can be found in Mendonça *et al.* [12].

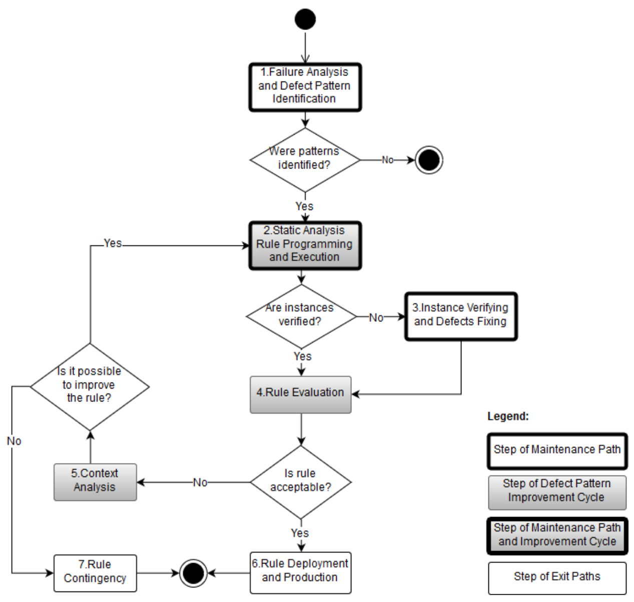

1.Failure Analysis and Defect Pattern Identification
Were patterns identified?
No
Yes
2.Static Analysis Rule Programming and Execution
Yes
Are instances verified?
No
3.Instance Verifying and Defects Fixing
Yes
4.Rule Evaluation
Is it possible to improve the rule?
No
5.Context Analysis
No
Is rule acceptable?
Yes
7.Rule Contingency
6.Rule Deployment and Production
Legend:
Step of Maintenance Path
Step of Defect Pattern Improvement Cycle
Step of Maintenance Path and Improvement Cycle
Step of Exit Paths

Figure 1: The PDM method [12]

Table 1: Defect Patterns Identified in Python/Django case [12]

Defect Pattern Name	Description	Defect Code Example	Fixed Code Example

Django ORM get	The application does not catch the exceptions thrown when a database search is conducted by id using Django ORM (Object-Relational-Mapper), and the id does not exist in the database.	`django.db.models import Model`
		`class Account(Model):`
		`    …`
		`...`
		`account =`
		`        Account.objects.get(id=id)`
		`...`
			`try:`
			`    account =`
			
			`Account.objects.get(id=id)`
			`except:`
			`   #handle the exception`

Float Conversion	The application does not catch the exceptions thrown when a string is converted to float.	`a = “217x"`
		`b = float(a)`
			`try:`
			`    b = float(a)`
			`except:`
			`    #handle the exception`

Date Conversion	The application does not catch the exceptions thrown when it converts a string to date.	`from datetime import datetime`
		`a = datetime.strptime(\`
		`     '10/10/201a',’%d/%m/%Y')`
			`try:`
			`    a = datetime.strptime(\`
			`          '10/10/201a', '%d/%m/%Y')`
			`except:`
			`   #handle the exception`

Table 2: Defect Patterns Identified in PHP case [12]

Defect Pattern Name	Description	Defect Code Example	Fixed Code Example

Date Conversion	A date conversion returns false when it fails. When a member function is called in a Boolean an exception is thrown.	$dt1 =
		\DateTimeImmutable::createFromFormat(
		'd/m/Y', $str1);
		$dt1 = $dt1->sub(new DateInterval('P1D'));
			…
			if(!$dt1){
			…
			}
			$dt1 = $dt1->sub(new DateInterval('P1D'));

Unchecked Integer	Data Access Object (DAO) layer may throw an exception when a non-validated integer variable is passed as parameter to their member functions.	$res = $someDao->someMethod($int_var);	if (strval($int_var) !=
			    strval(intval($int_var))){
			    …
			}
			$res = $someDao->someMethod($int_var);

Unchecked Id	Data Access Object (DAO) layer may throw an exception when a non-validated identifier variable is passed as parameter to their member functions.	$res = $someDao->someMethod($id_var);	if (!isset($id_var) ||
			    empty($id_var) ||
			    !is_numeric($id_var)){
			    …
			}
			$res = $someDao->someMethod($id_var);

2.2 Custom Rules

In this subsection, we briefly explain the cases were PDM was applied before. Those applications of the method produced the custom static analysis rules that were subjects for reuse.

In the first case, PDM was applied to a financial web application marketed as a software-as-a-service (SaaS) in Brazil since 2010. Although this software has been in use for eight years and a significant part of its defects had already been fixed, it was still being evolved, under active maintenance, and eventually presenting failures. Some of these failures were caused by errors of use of the service; however, the application should not raise unhandled exceptions in those situations.

The server-side of the application had 12 KLOC developed in Python with the Django Framework running on Apache HTTP Server. Table 1 presents the defect patterns that were identified in this application. SonarQube was being used to control the code quality of the software under study; therefore, it was selected as the static analysis tool for the method application. For each defect pattern was implemented a custom static analysis rules using SonarQube XPath plugin for custom rules in Python written software. PDM was used to improve the precision of rules achieving the level presented in Table 3. The relative recall [12] (see section 3.2) of all custom rules derived from the first application reached the level of 100% [12].

Figure 2 presents a simplified version of Django ORM Get Rule being executed in SonarQube SSLR Toolkit, an development environment that allow write and debug a rule in XPATH. In Figure 2, the source code under evaluation is presented in the left pane wheather its AST is presented in the right panel and the XPath rule in the botton one. In this simplified version, the rule uses method name (get) as well as the structure of the instruction, i.e., the use of two dot operators, to identify a call to the method not surrounded by a try/catch block. Notice that the class identifier is not fixed by the rule. As get method is from Django Framework, it would be usefull to reuse this rule for other software.

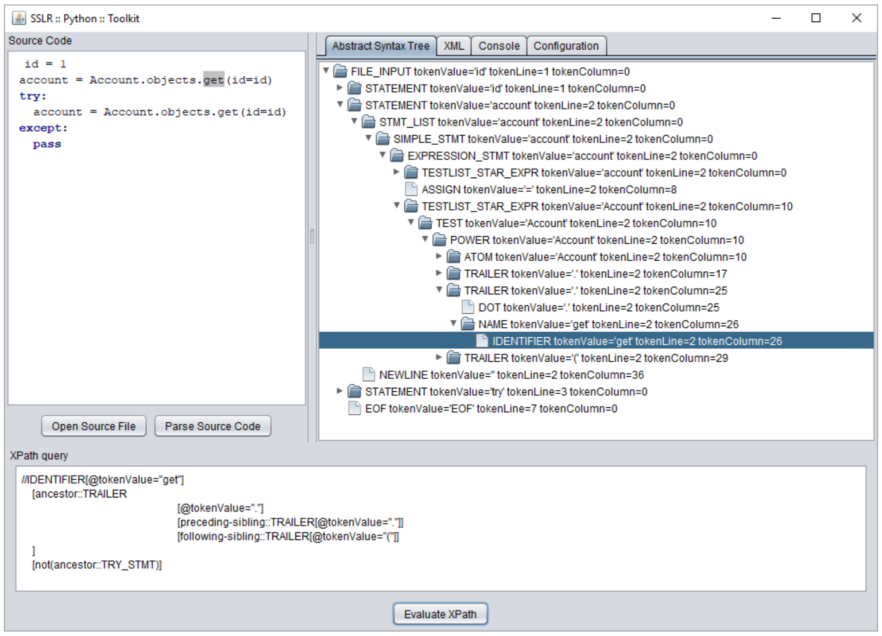
The second PDM application was a small-sized administrative web application of an educational institution, written in PHP. The failures caused by unhandled exceptions in this application were extracted from the web server logs. Table 2 presents the defect patterns identified [12]. This time the rules were programmed in Java instead of XPath since SonarQube's support for PHP custom rules is only available in Java. The precision of those rules is presented in Table 3. The relative recall of PHP rules was not evaluated since PDM was applied in a practical way, i.e., adjusting the rule for discarding false positives as soon as they are found [12]. Details of how PDM was applied to derive those rules can be found in Mendonça *et al.* [12].

Figure 3 presents a part of Unckecked Id PHP rule that checks whether a variable is inside a validation block, i.e., an try/catch or specific if instruction (see Table 2 for details of PHP rules). The source code of the rule give us some intuition about its reusability. Notice that specific part of the rules are fixed as string literals inside the rule source code. Adicionally, the structures verified by the rule code, i.e., if and try/catch blocks may be different in other software, such as ternady operator. For this reason, parts of the rule may need to be adjusted when reusing it. In other hand, the method presented in Figure3 is quite generic allowing those fixed parts to be modified with few effort.

Althought the source code of rules give us insights about its reusability, experimentation is necessary to have more concrete conclusions. The methodology used in our experimentation is presented in the next section.

Table 3: Precision of rules in their original application

Custom Rule	Language	Precision
Django ORM Get	Python/Django	68%
Date Conversion	Python/Django	59%
Float Conversion	Python/Django	67%
Date Conversion	PHP	100%
Unchecked Integer	PHP	100%
Unchecked Id	PHP	89.5%

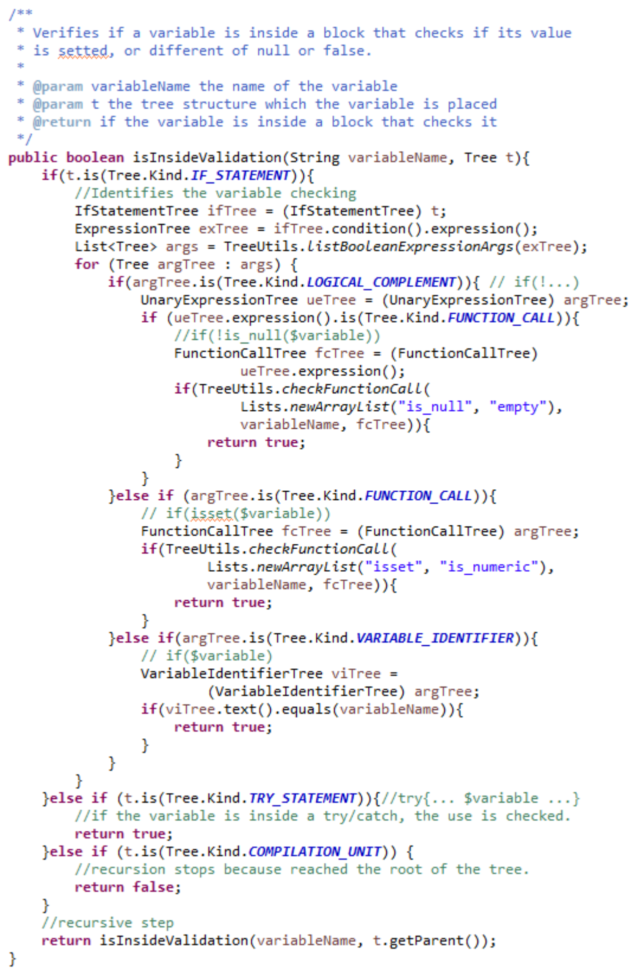


```
 id = 1
account = Account.objects.get(id=id)
try:
  account = Account.objects.get(id=id)
except:
  pass
```

```
▼ FILE_INPUT tokenValue='id' tokenLine=1 tokenColumn=0
  ▶ STATEMENT tokenValue='id' tokenLine=1 tokenColumn=0
  ▼ STATEMENT tokenValue='account' tokenLine=2 tokenColumn=0
    ▼ STMT_LIST tokenValue='account' tokenLine=2 tokenColumn=0
      ▼ SIMPLE_STMT tokenValue='account' tokenLine=2 tokenColumn=0
        ▼ EXPRESSION_STMT tokenValue='account' tokenLine=2 tokenColumn=0
          ▶ TESTLIST_STAR_EXPR tokenValue='account' tokenLine=2 tokenColumn=0
            ASSIGN tokenValue='=' tokenLine=2 tokenColumn=8
          ▼ TESTLIST_STAR_EXPR tokenValue='Account' tokenLine=2 tokenColumn=10
            ▼ TEST tokenValue='Account' tokenLine=2 tokenColumn=10
              ▼ POWER tokenValue='Account' tokenLine=2 tokenColumn=10
                ▶ ATOM tokenValue='Account' tokenLine=2 tokenColumn=10
                ▶ TRAILER tokenValue='.' tokenLine=2 tokenColumn=17
                ▼ TRAILER tokenValue='.' tokenLine=2 tokenColumn=25
                    DOT tokenValue='.' tokenLine=2 tokenColumn=25
                  ▼ NAME tokenValue='get' tokenLine=2 tokenColumn=26
                      IDENTIFIER tokenValue='get' tokenLine=2 tokenColumn=26
                ▶ TRAILER tokenValue='(' tokenLine=2 tokenColumn=29
      NEWLINE tokenValue='' tokenLine=2 tokenColumn=36
  ▶ STATEMENT tokenValue='try' tokenLine=3 tokenColumn=0
    EOF tokenValue='EOF' tokenLine=7 tokenColumn=0
```

```
//IDENTIFIER[@tokenValue="get"]
   [ancestor::TRAILER
                          [@tokenValue="."]
                          [preceding-sibling::TRAILER[@tokenValue="."]]
                          [following-sibling::TRAILER[@tokenValue="("]]
   ]
   [not(ancestor::TRY_STMT)]
```

Figure 2: A simplified version of Django ORM Get rule in SonarQube SSLR Toolkit



```
/**
 * Verifies if a variable is inside a block that checks if its value
 * is setted, or different of null or false.
 *
 * @param variableName the name of the variable
 * @param t the tree structure which the variable is placed
 * @return if the variable is inside a block that checks it
 */
public boolean isInsideValidation(String variableName, Tree t){
    if(t.is(Tree.Kind.IF_STATEMENT)){
        //Identifies the variable checking
        IfStatementTree ifTree = (IfStatementTree) t;
        ExpressionTree exTree = ifTree.condition().expression();
        List<Tree> args = TreeUtils.listBooleanExpressionArgs(exTree);
        for (Tree argTree : args) {
            if(argTree.is(Tree.Kind.LOGICAL_COMPLEMENT)){ // if(!...)
                UnaryExpressionTree ueTree = (UnaryExpressionTree) argTree;
                if (ueTree.expression().is(Tree.Kind.FUNCTION_CALL)){
                    //if(!is_null($variable))
                    FunctionCallTree fcTree = (FunctionCallTree)
                            ueTree.expression();
                    if(TreeUtils.checkFunctionCall(
                            Lists.newArrayList("is_null", "empty"),
                            variableName, fcTree)){
                        return true;
                    }
                }
            }else if (argTree.is(Tree.Kind.FUNCTION_CALL)){
                // if(isset($variable))
                FunctionCallTree fcTree = (FunctionCallTree) argTree;
                if(TreeUtils.checkFunctionCall(
                        Lists.newArrayList("isset", "is_numeric"),
                        variableName, fcTree)){
                    return true;
                }
            }else if(argTree.is(Tree.Kind.VARIABLE_IDENTIFIER)){
                // if($variable)
                VariableIdentifierTree viTree =
                        (VariableIdentifierTree) argTree;
                if(viTree.text().equals(variableName)){
                    return true;
                }
            }
        }
    }else if (t.is(Tree.Kind.TRY_STATEMENT)){//try{... $variable ...}
        //if the variable is inside a try/catch, the use is checked.
        return true;
    }else if (t.is(Tree.Kind.COMPILATION_UNIT)) {
        //recursion stops because reached the root of the tree.
        return false;
    }
    //recursive step
    return isInsideValidation(variableName, t.getParent());
}
```

Figure 3: Part of Unchecked Id PHP rule that verifies whether a variable is inside a validation block

3 Methodology

In order to assess the reusability of custom static analysis rules, we conducted a proof of concept by appling the rules explained in the previous section to similar projects. In this study, we aim to answer the following research questions:

RQ1. (sensitivity) In which scope custom rules can be reused?

RQ2. (effect) What are the benefits and effort of reusing custom rules?

RQ3. (sensitivity) Which factors have an impact on reusing custom rules?

In the next three subsections, we explain the software selection for our reuse investigation, the metrics used, and the process for reusing the rules.

3.1 Software Selection

As observed during rules' creation [12], the architecture of the applications plays an essential role in the definition of custom rules. Therefore, we informally expect custom rules produced in one software to be reusable in other software with similar architecture. As the intention of our study was to get the very first insights about reusing custom static analysis rules, we selected few projects and performed an in-depth analysis of reusing rules on them. An alternative approach could be sending pull requests for many open source projects asking for acceptance. However, as we previously did not know the precision and recall of reusing those rules, this approach was not appropriate since it could generate several false positives and vex project maintainers. Table 4 presentes a summary of software selected for reusing rules. Hereafter, we explain how those software were selected.

The first set of rules is for Python/Django written web applications, while the second is rules for PHP applications [12]. As Django is a popular framework that defines its reference architecture, we were able to find software projects with a similar architecture beyond the frontier of the company where the Python/Django rules were produced. Hence, we were able to conduct cross-company rule reuse evaluations for the rules created for Python/Django. In the case of the rules created for PHP, the architecture of the software used to produce the rules was specific and defined by the company. Therefore, we conducted a within-company rule reuse evaluation for these rules.

The software selection for the cross-company evaluation considered both, the use of similar technologies and the maturity of the project. We selected two software projects for the cross-company evaluation. The first one was an information system for undergraduate student's performance monitoring named CADD[1] (Student Performance Evaluation Commissions Support System). The server-side of the application had 4.8 KLOC written in Python/Django. The CADD system was developed over a one-year period by two CEFET/RJ[2] undergraduate students as a final course project. The testing of the CADD system was performed ad-hoc, without using a systematic procedure. The system tests presented several defects, thus reflecting a low level of maturity.

The second cross-company project was an open-source agile project management software named Taiga[3]. Taiga back-end[4] had 30 KLOC written in Python/Django within a history of four years of releases. This project is actively maintained and has over a thousand forks and five thousand stars on Github. Hence, we considered that Taiga had a higher level of software maturity than the CADD system.

As previously mentioned, the architecture of the software used to produce the second set of rules was defined by the company, and software with similar architecture was available for evaluation only within that company. Hence, we performed a within-company reuse evaluation for the PHP rules.

The software selected for within-company rules reuse evaluation was a 1.7 KLOC PHP written application with architecture similar to the software that originated the PHP rules. This application was recently developed over a four months period by three developers. Its purpose was to help the employees of the company to register themselves in the new corporative email

[1] https://github.com/diogosmendonca/CADD
[2] http://www.cefet-rj.br
[3] https://taiga.io/
[4] https://github.com/taigaio/taiga-back

system. Hereafter we refer to this application as Registration system.

Table 4: Software selected for reuse of rules

Software	Technology	Size (KLOC)	License Type
Registration	PHP	1.7	Proprietary
CADD	Python/Django	4.8	Open Source
Taiga	Python/Django	30	Open Source

3.2 Reuse Process

The reuse of rules may affect their precision and recall. In this way, to properly reuse custom static analysis rules, a systematic approach to verify their precision and relative recall [12] is needed. However, for precise verification of both metrics, an extensive inspection of the software to which the rules are to be applied may be required, thus hindering possible effort reduction benefits of reusing rules. For solving these conflicting requirements, we propose the following four-step process to reuse rules and to perform their verification.

1. Produce and run a relaxed version of the rule for locating defect candidates. This relaxed version must be broader than the original rule, ideally finding all possible locations in which the defects to be found can be present. For example, if a rule searches for a possible defect in a specific function call, a relaxed version of the rule could be one finding all calls of this function. This step can be supported by using regular expressions or an IDE's search feature. The intend of this relaxed version is to prevent rule overfitting by verifing the rule against false negatives, an activity that is covered in step 3.
2. Run the custom static analysis rule on the new software to be verified.
3. Start to inspect the negative cases, i.e., the defect candidates found in step 1 but not alerted by the rule executed in step 2. If a false negative is found, stop the inspection, modify the rule to also alert the false negative case found, and go back to step 2. When no false negative is found, the relative recall is 100% and we can move forward for precision verification in step 4. In step 3, the number of inspection points might be high. To reduce the inspection effort, depending on the confidence that a rule engineer has on the rule, she may choose to inspect a sample of the negative cases to confirm whether the rule has false negatives, or even to not inspect them. However, such an approach does not allow to calculate the relative recall and introduces the risk of having false negatives.
4. Inspect the alerts produced by the rule for calculating its precision.

The proposed process intends to delimitate the inspection scope, reduce the inspection effort, and to assure a relative recall of 100% in the reused rules. The inspection scope is limited by the relaxed version of the rule produced in step 1. The iterative and incremental nature of the process aims to reduce the inspection effort by eliminating defect candidates when eagerly changing the rule during the process. If the developer correctly implemented the changes in the rule to remove false negatives, the relative recall will achieve 100% by design. Finally, after applying the reuse process, the precision improvement of a rule can be performed by applying PDM's improvement cycles [12].

3.3 Metrics

Table 5: Metrics used in the reuse of rules evaluation.

Precision	$\frac{Defects\ alerted}{Defects\ alerted + False\ positives\ alerted}$
Relative Recall	$\frac{Defects\ alerted}{Defects\ matching\ the\ intended\ rule}$
Inspection Reduction Potential	$1 - \frac{Alerts\ produced\ by\ the\ rule}{Alerts\ of\ relaxed\ version\ of\ rule}$

To answer our research questions, we used the set of metrics presented in Table 5. It is noteworthy that the recall is relative to the defects matching the intended rule [12]. Those are located by inspecting or testing the candidate defects retrieved by the relaxed version of the rule. This version should be broad enough to retrieve all defects but might include false positives. We introduce a new metric called inspection reduction potential, which measures the percentage of inspection points that in the future could be avoided from being inspected by reusing the rule. The inspection reduction potential metric assumes that the ratio between negative cases discarded and alerts found during the reuse of a rule will be maintained in the future. In this way, the negative cases will not be inspected anymore. The inspection reduction potential measures the potential benefit of reusing a rule, considering the alternative as searching for the defects using a relaxed rule version (e.g., an IDE's search feature) and inspecting retrieved instances manually.

4 Results

In this section, we present the results of reusing a set of rules in two different contexts, cross-company and within-company.

4.1 Cross-company rules reuse evaluation

In this subsection, we explain the two cross-company rule reuse evaluations, i.e., evaluations on reusing the rules beyond the context of the company in which they were created. The first case concerns a software system from a different company, and the second one concerns an open-source software system. For these evaluations, we used the Python/Django rules implemented to reveal the defect patterns shown in Table 1. 4.1.1 CADD system

As evaluation procedure for the reuse of Python/Django rules, we executed them for the CADD system and tested the alerts produced for defect confirmation. The relaxed versions of rules were defined to find the function calls present in each pattern presented in Table 1. Table 5 presents the results of applying the reused rules. Further details on how those rules were produced can be found in [12].

The CADD system did not have any float or strptime function calls, thus float conversion and date conversion rules were not applicable. Regarding the Django ORM get rule, we found 64 instances of the get function call. All function calls not alerted by the rule were inspected for confirming that the static analysis rule works correctly. After confirming a relative recall of 100%, the calculated inspection reduction effort was 62.5% (see Table 5).

The function calls alerted by Django ORM get pattern were tested for defect confirmation. The precision level of 75% found is slightly superior to the precision found in the software in which the rule was produced (68%). We inspected the false positives of the CADD system for causal analysis and found similar contexts causing the pattern to fail from the ones in the software which originated the rule. As observed during the elaboration of the rules [12], this precision could have been further improved by using a tool with more resources for rule programming than SonarQube.

The level of precision found (75%), and the number of defects discovered in the software (18) strengthen our confidence that custom rules may be reused in a cross-company setup to find defects in less mature software and to reduce future efforts of inspection. With the intent to help researchers and practitioners to better understand and check our results, we made the artifacts used in our evaluation available online [5].

4.1.2 Taiga

Taiga is a 30 KLOC, Python/Django written software with several installations and users. We chose Taiga to evaluate rule reuse in a more mature software than the CADD system.

As the evaluation procedure, we also executed the same Python/Django rules on Taiga, excluding automated tests and migrations (database creation scripts) from the analysis. We also searched for the function calls present in the rules for checking if the rules were working correctly. Table 5 presents the number of function calls and alerts found. After inspecting approximately 50% of the alerts without finding any defects, we found three new fixing alternatives that properly prevent the false positive defect candidates alerted by the Django ORM get rule. Those contexts are explained in Table 6.

After finding these contexts, we realized that a new PDM defect pattern improvement cycle would have to be performed to reuse Django ORM get rule in Taiga effectively. As some of these contexts could be very complicated or even impossible to include in the Django ORM get rule using SonarQube (Figure 2 presents a simplified version of this rule) we decided not to perform the improvement cycle. Furthermore, the effort to continue inspecting Taiga without an expectation of executing an improvement cycle of the rule would not be worthwhile for the study purpose. Thus, we decided not to continue inspecting Taiga and finished the evaluation.

We conclude from the Taiga evaluation that custom rules may not be reusable in other software without adaptation, even when both software systems use the same framework or reference architecture. Programming style and architecture could be different from one software to the other, and the execution of a PDM defect pattern improvement cycles may be needed. Furthermore, as also observed in the CADD system, the use of previously defined custom rule may reduce the effort of checking a system for a specific defect. In the case of Taiga, the Django ORM get rule execution enabled to reduce the inspection effort, discarding the need of inspecting 81 out of the 139 ORM get function calls, representing inspection reduction potential of 58% (naive estimate considering all function calls to require the same effort). The not inspected function calls represents the ones discarded by rule. We had confidence not to inspect those function calls since Django ORM Get rule was successfully applied before in two software systems with 100% of relative recall.

4.2 Within-company rules reuse evaluation

As an evaluation procedure, we started by executing the three previously PHP rules, produced to reveal the patterns shown in Table 2, without any modification on the Registration system. In the first execution, no defect candidate was found by the rules. With the intent of verifying this result, with the aid of an IDE, we searched in the source code for the elements contained in each defect pattern. We did not find any createFromFormat function call or integer variables being passed to the DAO layer, which were the elements of two of the rules (date conversion and the unchecked integer patterns, respectively). However, we have found many id variables being passed to the DAO layer, which are the elements of the remaining rule (unchecked id). The rule was not able to find those function calls because the naming convention for the DAO instance variables (architectural relevant element) changed from the original system to the Registration system. Tarhus the rule was adjusted to reflect the new naming convention, and was again executed against the Registration system. Table 5 presents the results of the adjusted rule execution and inspection together with the total number of all function calls.

The adjusted unchecked id rule found five defect candidates in a total of 50 function calls, which represents an inspection reduction potential of 90%. We inspected the alerts produced, and two of them were confirmed as defects, thus reflecting a precision of rule of 40%. Although this precision is low, the company decided to use the rules in its production environment for the Registration system. This decision was influenced by the positive result obtained in the first experiment with those rules, which strengthened the confidence of the company practitioners that the rules are useful for finding defects. Additionally, the low absolute number of false positives (three) associated with the functionality

[5] https://github.com/diogosmendonca/CADD/issues/1

of SonarQube of marking false positive alerts not to be shown within the developer’s IDE made the effect of false positives irrelevant for the developers.

We conclude from the Registration system evaluation that it is possible to reuse custom rules in a within-company environment. We also found that the adoption of the rules, in this case, was influenced by the previous experience of the company with the rules and that the precision may have had low influence on this decision.

5 Discussion

In this section, we discuss each research question based on the results of our study.

RQ1. (sensitivity) In which scope can custom rules be reused?

Our evaluations indicate that custom rules might be reused in other software in within-company and cross-company environments. Table 5 shows a summary of our quantitative results. The defects found showed a potential positive reuse effect of custom rules on web application reliability. This potential can be achieved not only in the maintenance and evolution phases but also in the software development phase. The CADD system was developed recently and, at the time of the writing of this document, it was not in production. The reuse of custom rules produced based on other software helped to identify several defects in the CADD system before it was released to its customers.

RQ2. (effect) What are the benefits and effort of reusing custom rules?

The beneficial effects of the reuse of custom rules are finding defects and reducing the number of defect candidates for verifying a defect pattern in other software. Table 5 presents our results. The precision of reused rules ranged from 40-75% without compromising the relative recall of 100%, in the cases for which this metric was measured. The inspection reduction potential for verification ranged from 58-90% in our studies, which might represent a significant effort reduction in verifying the presence of a defect pattern in a software system.

The effort of the overall approach for reusing rules depends on the project and how the reuse of rules is applied. The worst case was found in Taiga evaluation. Despite the fact that reusing rules could discard 58% of the alerts, the remaining 42% (58 alerts) would still need to be inspected. During the inspection of the alerts the recommendation for reducing its effort is to, as soon as a new context, i.e., a new structure that avoids the defect, is found the static analysis rule should be modified to discard all instances of it at once. However, in Taiga we found some contexts that are extremely difficult to include in static analysis rules, thus we needed to manually inspect the remaining alerts (we took a part-time week inspecting and stopped because of time constraints). The effort in this case (worst one) is proportional to the effort of inspecting remaining alerts and modifying the rule after applying it.

On the other hand, there are cases in which reusing a rule is very simple and straightforward. In the Registration system, 90% of the alerts were discarded remaining only 5 to be inspected. A simple adjustment of naming convention was done in the rule. The rule was successfully reused in a few hours (less than 4 hours). We found 2 defects, guaranteeing that in the software there is no other similar defect and introduced an automated verification against reinfection with this effort. In all cases, the relaxed version of the rule used was an IDE search, which is quite fast to produce, i.e., produced in minutes.

RQ3. (sensitivity) Which factors have an impact on reusing custom rules?

We found some influence factors for custom rules reuse and adoption. First, architecture plays an essential role in rules definitions and, consequently, in its reuse. Architectural elements such as naming convetions and use of layers were relevant for reuse of rules. However, architecture similarity is not enough for rule reuse. As we observed in the Taiga and Registration systems, differences in the way that architecture is implemented and programming style might cause rules not to work correctly. Hence, the adjustment of the rules might be needed to enable effective reuse. An influence factor for reused rules adoption in a within-company environment is the success of rules in finding defects on other software. Indeed, the influence of this factor overcame the low precision of rules achieved in the Registration system, and the company chose to deploy the rule for defect prevention.

Table 5: Evaluation of reused rules

System	Type of Reuse	Technology	Rule	Function Calls (relaxed rule)	Alerts produced by the rule	Defects found	Precision	Relative Recall	Inspection Reduction Potential
CADD	Cross-company	Python/ Django	Django ORM Get	64	24	18	75%	100%	62.5%
Taiga	Cross-company	Python/ Django	Django ORM Get	139	58	0	N/A	N/A	58%
Registration	Within-company	PHP	Unchecked Id	50	5	2	40%	N/A	90%

Table 6: New contexts found for Django ORM get rule in Taiga

Context Description	Code Example
The use of pk to access an identifier attribute passed to get method instead of an id attribute	`User.objects.get(id=otherObject.pk)`

Id validation using a validator and inheritance. ProjectExistsValidator checks if the project exists and is called through inheritance on DueDatesCreationValidator is_valid method.	`class ProjectExistsValidator:`
	`    def validate_project_id(self, attrs, source):`
	`        …`
	
	`Class DueDatesCreationValidator(`
	`           ProjectExistsValidator, validators.Validator):`
	`    project_id = serializers.IntegerField()`
	`     …`
	
	`validator = validators.DueDatesCreationValidator(`
	`data=request.DATA, context=context)`
	
	`if not validator.is_valid():`
	`           return response.BadRequest(validator.errors)`
	
	`project_id = request.DATA.get('project_id')`
	`project = models.Project.objects.get(id=project_id)`

Constant as a literal or attribute.	`class BaseEventHook:`
	`    platform = "Unknown"`
	`    …`
	`    def get_user(self, user_id, platform):`
	`     …`
	`        user = get_user_model().objects.get(`
	`                  is_system=True,`
	`                  username__startswith=platform)`

6 Threats to Validity

Internal Validity. The verifications of the results produced by the rules were conducted by a single researcher. However, the artifacts used in CADD system evaluation are available online[5], allowing the investigations to be replicated by others to confirm the obtained results. Regarding the artifacts of the Registration System and Taiga, they were not publicly released. The Registration System is proprietary, and its artifacts could not be released. In the case of Taiga, the artifacts were not produced with evidence (print screens), as they were produced in the CADD System, so we decided not to publish them. Additionally, the partial inspection of defect candidates, with the purpose of verifying the correctly working of a rule during its reuse might have caused missing false negatives. Inspecting all defect candidates discarded by a rule in some cases may not be a practical solution for confirming if it is working correctly. One of the expected benefits of reuse of rules is reducing the effort of inspection, and by inspecting all defect candidates this benefit would be achieved only in the future after reusing the rule.

Construct Validity. We selected the software systems for the study by convenience. For instance, the selection of software that was developed by students, who have a novice level of experience in software development, might have influenced the evaluation of the reusability of the rules. It is noteworthy that the defects detected by the patterns in our study are more commonly introduced by novice developers than by experienced ones.

Conclusion Validity. The amount of systems chosen for evaluation does not allow applying any more sophisticated statistical techniques. Instead of claiming for conclusion validity, we addressed the research questions using a qualitative approach, trying to gather an initial understanding of the reuse scope, effects, and factors.

External Validity. We recognize that the evaluations and results presented in this section are only examples of the reuse of rules produced by applying PDM. The quantitative results achieved cannot be extrapolated for any software other than the ones in which the evaluations were performed. Thus, our findings should be interpreted as preliminary results from a specific context.

7 Related Work

In this section, we review other works on the reuse of static analysis rules.

Shekhovtsov *et al.* [10] proposed a conceptual model to trace back static analysis rules to design decisions. In this way, the reuse of rules could occur by identifying similar design decisions, shared between applications, to then select the rules to be reused. Despite the conceptual model, the authors do not provide any experimental results of its application.

There is research on reusing static analysis between DSLs [16] and between languages [17], working at a higher level of abstraction for reuse. Our work differs from them in its purpose. Instead of aiming cross-language reuse, our study aims to evaluate the practical reuse of custom rules in other projects with the same programming language. In this way, we try to reuse the rules as they were developed, evaluate them and make adjustments if needed.

Gurgel *et al.* [11] proposed a DSL called TamDera that allows the reuse of static analysis rules in an object-oriented way, i.e., by extending an existent rule. TamDera was designed with the purpose of checking architecture against architectural drift and erosion symptoms. The reuse of TamDera written rules was verified in an experiment involving 21 versions pertaining to 5 software projects, and more than 600 rules. The authors were able to reuse 72% of the rules; however, the precision of reused rules was not provided, neither measures of the possible benefits of reusing those rules, such as reduction of the number of inspection points.

To the best of our knowledge, our study is the first to report the precision and relative recall when reusing custom static analysis rules. We also introduced and are the first to measure the reduction inspection potential caused by the reuse of custom rules. The measures and experiments performed used tools that are commonly available and extensively used in industry, such as SonarQube[4], being a practical approach for reuse and allowing other researchers to replicate our studies in different contexts.

8 Concluding Remarks

We found that custom static analysis rules can be reused in within- or cross-company environments and not only for software in the maintenance phase but also recently developed ones. We were able to find defects in other software by reusing custom rules, as well as to reduce the verification effort of defect patterns. Nevertheless, the architecture and programming style played an essential role in successfully reusing custom static analysis rules, thus being an influencing factor for reuse. Our work also would help developers to identify cases were the rules would not be reusable and to avoid them.

In this way, the reuse of rules has the advantage of potentially producing more robust rules and reducing the effort of identifying similar patterns in other systems. We also observed that previous successful experience with custom rules might influence rule reuse adoption.

Based on our experience, we recommend some practices for the evaluation and implementation of reusing custom rules. After executing a rule in another software, our advice is to inspect both, the alerts produced and other potential defect candidates that were not alerted. The inspection of the former might show new contexts to include in the rule to avoid false positives, and the latter might present adjustable cases where the rules fail because of differences in the architecture implementation or programming style. After inspecting these cases, fully or incrementally, the rules can be adjusted and executed to identify defects. Furthermore, additional PDM defect rule improvement cycles can also be performed to improve the precision of the rules if needed.

ACKNOWLEDGMENTS

We specially thank Professor Arndt von Staa for his contribution in the beginning of this research with his insights and ideas. We also thank the system analysts of DTINF/CEFET-RJ for supporting this work. This work was partially supported by the CNPq grant 141345/2015-2.

REFERENCES

[1] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Inf. Softw. Technol.*, vol. 53, no. 4, pp. 363–387, 2011.

[2] T. Muske and A. Serebrenik, "Survey of approaches for handling static analysis alarms," in *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*, 2016, pp. 157–166.

[3] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using Static Analysis to Find Bugs," *IEEE Softw.*, vol. 25, no. 5, pp. 22–29, 2008, doi: 10.1109/MS.2008.130.

[4] SonarSource, "SonarQube," 2008. [Online]. Available: https://www.sonarqube.org/. [Accessed: 20-Jul-2018].

[5] InfoEther Inc, "PMD." [Online]. Available: http://pmd.github.io/.

[6] The University of Maryland, "FindBugs." [Online]. Available: http://findbugs.sourceforge.net/. [Accessed: 01-Jul-2016].

[7] Y. Tymchuk, M. Ghafari, and O. Nierstrasz, "JIT feedback: What experienced developers like about static analysis," in *Proceedings - International Conference on Software Engineering*, 2018, doi: 10.1145/3196321.3196327.

[8] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, vol. 1, pp. 470–481, doi: 10.1109/SANER.2016.105.

[9] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in *ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, doi: 10.1145/2970276.297.

[10] V. A. Shekhovtsov, Y. Tomilko, and M. D. Godlevskiy, "Facilitating Reuse of Code Checking Rules in Static Code Analysis," in *Lecture Notes in Business Information Processing*, 2009, pp. 91–102.

[11] A. Gurgel *et al.*, "Blending and reusing rules for architectural degradation prevention," in *Proceedings of the 13th international conference on Modularity - MODULARITY '14*, 2014, doi: 10.1145/2577080.2577087.

[12] D. S. Mendonça *et al.*, "Applying Pattern-driven Maintenance: A Method to Prevent Latent Unhandled Exceptions in Web Applications," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2018, pp. 31:1--31:10, doi: 10.1145/3239235.3268924.

[13] A. Bessey *et al.*, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, 2010, doi: 10.1145/1646353.1646374.

[14] M. Kalinowski, D. N. Card, and G. H. Travassos, "Evidence-Based Guidelines to Defect Causal Analysis," *IEEE Softw.*, vol. 29, no. 4, pp. 16–18, Jul. 2012, doi: 10.1109/MS.2012.72.

[15] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 672–681, doi: 10.1109/ICSE.2013.6606613.

[16] J. Mey, T. Kühn, R. Schöne, and U. Assmann, "Reusing Static Analysis across Different Domain-Specific Languages using Reference Attribute Grammars," *Art, Sci. Eng. Program.*, vol. 4, no. 3, Feb. 2020, doi: 10.22152/programming-journal.org/2020/4/15.

[17] D. Darais, M. Might, and D. Van Horn, "Galois transformers and modular abstract interpreters reusable metatheory for program analysis," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, 2015, doi: 10.1145/2814270.2814308.