Renraku — the One Static Analysis Model to Rule Them All

Yuriy Tymchuk
SCG @ University of Bern
Switzerland

Mohammad Ghafari
SCG @ University of Bern
Switzerland

Oscar Nierstrasz
SCG @ University of Bern
Switzerland

Abstract

Most static analyzers are monolithic applications that define their own ways to analyze source code and present the results. Therefore aggregating multiple static analyzers into a single tool or integrating a new analyzer into existing tools requires a significant amount of effort.

Over the last few years, we cultivated Renraku — a static analysis model that acts as a mediator between the static analyzers and the tools that present the reports. When used by both analysis and tool developers, this single quality model can reduce the cost to both introduce a new type of analysis to existing tools and create a tool that relies on existing analyzers.

CCS Concepts • **Software and its engineering → Software maintenance tools**; *Extra-functional properties*; Object oriented architectures; Abstraction, modeling and modularity;

Keywords static analysis, code quality, software design

ACM Reference format:
Yuriy Tymchuk, Mohammad Ghafari, and Oscar Nierstrasz. 2017. Renraku — the One Static Analysis Model to Rule Them All. In *Proceedings of IWST '17, Maribor, Slovenia, September 4–8, 2018,* 10 pages.
https://doi.org/10.1145/3139903.3139919

1 Introduction

Smalltalk has a long history of development tools that aid programmers to perform their daily tasks. Some of these tools use static analysis to reveal additional information about source code to a developer. Tools such as SmallLint quality checker, Refactoring Browser and Rewrite Engine [8] played a crucial role in software maintainability. On the other hand, tools that aid to maintain code have their own maintainability cost. In the last decade there were several attempts to harvest useful information about source code properties. Such information includes test coverage [3], usage contracts adherence [7], code churn, exception stack traces, and IDE interactions [5]. While all the properties may convey useful information, it cannot be accessed from the development tools. It is expensive to maintain integration of multiple data retrieval engines in various development frontends. Even if we consider only SmallLint — the de-facto static analysis engine of Smalltalk, there is only a single standalone tool (RefactoringBrowser or CriticBrowser) per Smalltalk dialect that fully supports SmallLint features. While there are other users of SmallLint reports such as Continuous Integration (CI) build jobs, they are rather basic as it is expensive to dig into the SmallLint architecture and maintain compatibility with future versions.

We hypothesize that a single model that provides information about various source code properties may facilitate the development of analysis engines and their integration into development tools. The main goal of such model is to decouple code analyzers and development tools and thus to reduce the cost of:

1. providing custom source code analysis reports in existing development tools;
2. obtaining and reusing the source code properties provided by available analyzers.

In this paper we describe Renraku [12] — the unified code quality model of Pharo[1] [6]. Renraku was shaped by 3 years of tool and analysis development, as well as user feedback. The word "Renraku" is Japanese (連絡) and means *communication*, *connection*, *coordination*. Renraku is mainly used by the live static analysis feedback system called QualityAssistant [11], yet other tools such as CriticBrowser, Calypso[2] code editor, and ViDI inspector [14] also rely on the same model. Renraku's main source of code properties comes from SmallLint. However, there were experiments of issue tracker entries integration as well as tests binding and code coverage assurance. While Renraku is still mainly used to provide linting static analysis reports for a live feedback tool, there are other users of this model that motivate the robustness and usefulness of Renraku as a generic code property bridge.

The paper is structured the following way: in Section 2 we provide a brief overview of the related work; Section 3 describes the fundamental idea of Renraku; the three main building blocks of Renraku are described in detail in Section 4,

IWST '17, September 4–8, 2018, Maribor, Slovenia

ACM ISBN 978-1-4503-5554-4/17/09…$15.00
https://doi.org/10.1145/3139903.3139919

[1] http://pharo.org
[2] https://github.com/dionisiydk/Calypso

Section 5, and Section 6; Section 7 describes the differences and the simmilarities of SmallLint and Renraku; Section 8 and Section 9 demonstrate how to use Renraku to create a quality rule and a quality-aware tool respectively; we mention most notable Renraku users in Section 10; Section 11 concludes the paper.

2 Related Work

The code quality rules present in Pharo are similar to those provided by the most popular static analyzers for other languages, such as FindBugs [1], Pylint[3], JSHint[4]. These analyzers provide a limited support to define new rules and ourput the validation results in a plain text. Renraku is designed to be an object-oriented framework that expects that all the requirements such as defining rules, running them and processing the output are going to follow object-oriented approaches. Thus a developer will subclass a basic rule class and specialize it upon a new rule creation. The quality reports will be actual objects as well and will have an extensible way to provide feedback or even define special behavior.

Google introduced Tricorder — a static analysis integrated into their pre-commit review [9]. Tricorder relies on 16 static analysis tools that can be applied to five programming languages. Besides having to integrate the output into their code review tool and invent a strategy to handle false positives, Google engineers had to build a sophisticated infrastructure to run all the tools on their codebase and provide a uniform result. Buckers *et al.* operated on a much smaller scale by running 3 static analysis tools on a single Java project and visualizing the obtained result [4]. And while the main focus of the authors is a tool that displays visualization of static analysis, they spend a large amount of time explaining the design decisions used to run all the analysis together and unify results. Because of the current design of static analysis tools, developers have to spend substantial amount of time to run the analysis and aggregate the reports, while their main goal is to incorporate static analysis feedback in a tool that they develop. With Renraku we propose a unified model for static analysis reports, thus a tool developer has to rely on a single set of API to work with static analysis.

On the other hand, if one wants to develop a new static analysis algorithm there is no standard way to do this, and as a result the author of the algorithm has to also develop and maintain the reporting mechanism for static analysis results. This happened to the *uContracts* Domain Specific Language (DSL) and validation engine which was developed to check the most common software design constraints [7]. Together with the analysis engine the authors had to develop a code editor plugin to run the constraint checking and report detected violations. uContracts was never used for real tasks because the code editor was replaced by a successor which worked with a different kind of plugin and the authors of uContracts did not have time to develop one more plugin for yet another editor. With Renraku we provide a unified model that code analyzers can use to provide their feedback and the tools will pick up and display the information automatically.

[3] https://www.pylint.org

[4] http://jshint.com

3 The Quality Triad

Renraku is based on three basic concepts as depicted in Figure 1.

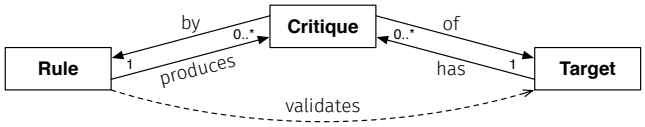

Figure 1. The Quality Triad of Renraku

A **critique** is a single report about code quality. It targets a single code entity and it is based on a single quality rule. A critique is the main unit that should be used to communicate code quality information to a user. Critiques can be specialized to provide a sophisticated explanation, solution suggestions, custom tooling for problem resolution and much more. A critique does not have to be negative, it can just be a link between a rule and a target that may have information of a different kind. In this case a system may contain all the possible links between all the targets and all the rules. Then a link can be re-evaluated in case the target or the rule were changed. On the other hand there might be multiple critiques connecting the same rule and target in case the target violates the rule multiple times. For example a class may have multiple unused instance variables and each critique will target a unique instance variable from the same class.

A **rule** defines a quality issue, it can identify an issue in a software entity and produce a critiques about it. Potentially a rule can produce multiple critiques about various targets. A rule can be viewed as a function that accepts an entity and returns a critique of it. Although a rule interacts with a target during the validation process, it does not store any direct references to the target and thus does not have any strong dependencies. A rule is also responsible for choosing a critique that is the most appropriate one for communicating an issue. For example if a visualization is needed to identify the cause of an issue the rule should use a critique capable of displaying visualizations.

A **target** is the actual piece of code that a critique targets. Potentially a target can be criticized by many critiques produced by various rules. A target should provide an interface to query its quality *i.e.,* return the critiques produced by available rules about it. The main reason for this functionality is to simplify the critique query process for potential tools. For example, when the developer of a code editor wants to add a quality feedback to his tool, obtaining the quality information should be as easy as asking the method or class itself what are

its critiques. This does not have to be the only way to obtain critiques about a target, but the simplicity of operation is important for the adoption of quality feedback in development tools.

While the Renraku triad envisions three main entities and their purpose, the system that we have built in reality is much more complicated as can be seen in Figure 2. In the rest of this paper we are going to discuss the decisions taken and the pitfalls that we encountered while implementing the Renraku model.

4 The Critique

According to the main Renraku vision, a critique should link a quality rule to a source code target and communicate the issue discovered by the rule. In our implementation, we also considered other kinds of reports not related to quality rules. Source code can have diverse sources of related data such as code review discussions, bug reports, test coverage, or even plain text notes. All the mentioned data sources may have the same or even higher importance than the static analysis feedback, and not every type of report is going to have some kind of a rule associated with it. We introduced `Property` — a superclass of `Critique` and other possible external properties related to a piece of source code. `Property` defines a basic interface of a title and an icon that can be used to display it in a user interface. Then `Critique` specializes `Property` by extracting the title from the rule name, selecting the icon based on a rule severity and additionally provides a description based in the rule's rationale. Currently there are not many properties used used in practice and thus in the context of this paper we mostly focus on critiques.

4.1 Source Anchors

While the ideal vision of Renraku suggests that a critique points directly to a target that violates the rule, in reality our targets are source code entities that have text as their main representation. Thus a critique should also specify which code interval violates the rule. For this reason a critique points to a source anchor, which knows about the target entity and the source code interval as can be seen in Figure 2.

```
1 relationGraphOnReverse: anObject
2    relationGraph := anObject.
3    self relationGraph build.
4    self buildReverseRoots
```

Listing 1. A method sends a message with a selector that no methods in the system implement.

```
1 check: aMethod
2    aMethod messages do: [ :selector |
3      (SystemNavigation allImplementorsOf: selector)
4         ifEmpty: [ self critiqueFor: selector ] ]
```

Listing 2. Rule validating a method for sending messages of unimplemented methods

Consider Listing 1 which presents a method extracted from one of Smalltalk frameworks. The system has no method with the selector `buildReverseRoots` and thus it is highly suspicious that this method sends such message. There are different approaches to identify which exact message has a selector that is not implemented by any method. One can traverse all the AST nodes, and check whether a node is a message send and whether it has a selector which does not match any method in the system. Then the rule will have a violating AST node which knows its interval in the source code. This is a good use case for a concrete source anchor that simply stores the interval itself. However AST traversal is time consuming and some AST nodes may be in the end optimized and replaced by a special bytecode. The actual rule implementation checks only the messages available directly in the bytecode as is shown in Listing 2. As the result the rule only knows which message violates it, but does not know the message's positions in the source code. For such cases there is a source anchor that derives the interval based on the substring location in the entitiy source code. Needless to say, the substring approach may be not precise given multiple occurrences of the same substring.

For class-based rules a developer can rely only on substrings, as a class definition has no AST representation and does not provide a way to easily access the position of the building blocks such as variable declaration, trait composition, *etc.* An example of a class definition is demonstrated in Listing 3. The `class` variable is unused and when detecting its interval in the source code, the first substring match occurs on the `subclass:` keyword. Such a report will confuse the developer so to mitigate the matching issue a dedicated variable-matching source anchor uses special heuristics (such that the variable name has to be surrounded by a space or quote) to identify the interval with a higher precision. It is up to a rule and critique developer to use available source anchors, or to create special ones to provide the best feedback possible with the critique and source anchor combination. One could argue that source anchors could be rendered obsolete if in the future all the rules would be based on AST nodes. We believe that as long as the program implementation is mainly defined as text, the source anchors are going to be important, because some rules may check textual features. For example one of the rules checks if the line breaks are encoded with the `cr` control character, and there is no way to analyze this on the AST level.

```
1 Model subclass: #RBClassToRename
2    instanceVariableNames: 'rewriteRule class'
3    package: 'Refactoring-Tests-Core-Data'
```

Listing 3. A class with an unused instance variable `class`

We discovered that for many critique titles it is enough to just show the name of the rule that produced the critique without significant changes. For the rest of critiques usually it is enough to add a couple of words to reasonably improve the

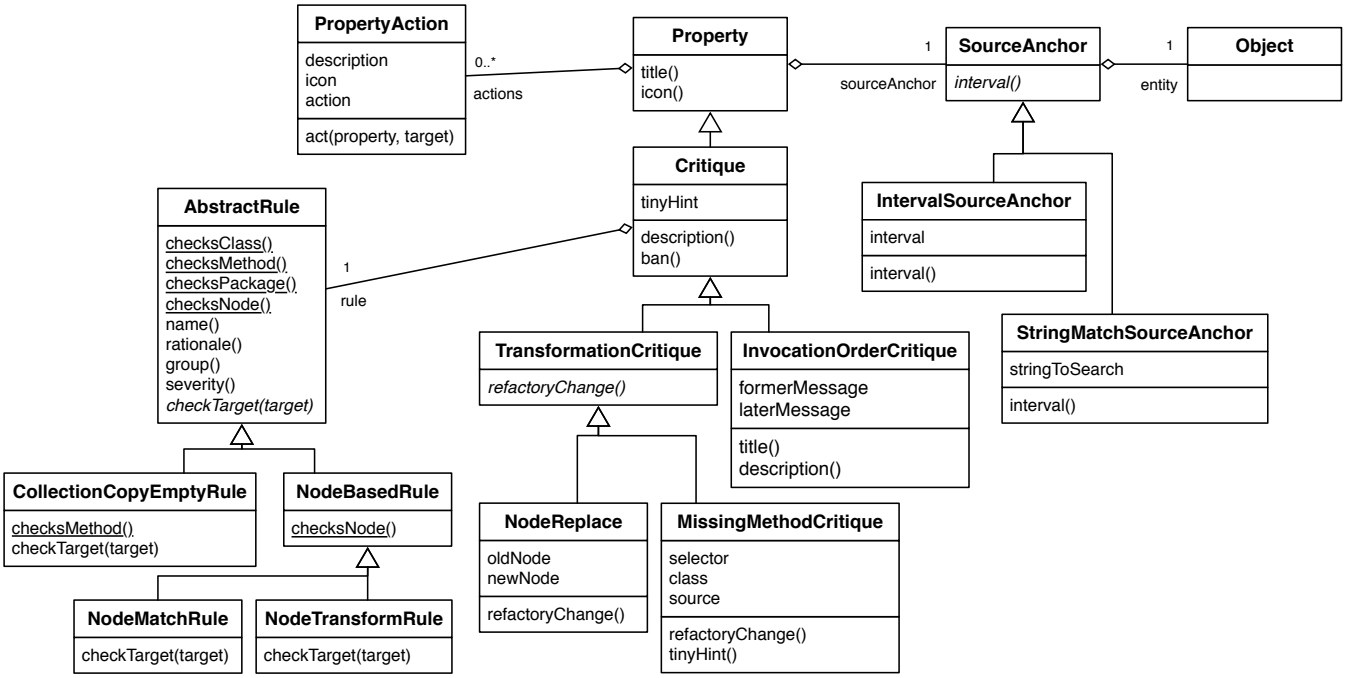

Figure 2. UML diagram of the main Renraku components

explanation of a critique for a particular case that it addresses. For this reason we introduced a *tiny hint* property of the critique that is represented by a short string and appears at the beginning if the title before the rule's name. For example, the title of a critique about an unused variable will look like this:

> [count] Instance variable neither read nor written.

In this case *count* is the variable's name, and the rest of the title is the rule's name. The tiny hint allows a developer to quickly identify the problematic piece of code while the rule's title briefly explains the issue.

4.2 Custom Actions

Another challenge of a good critique model design arises when rule developers need a flexible approach to provide a custom behavior to their client while staying tool-agnostic. For example a common aid provided by quality violations before Renraku was to transform the detected issue with rewrite expressions. To support this, the quality-aware tools had to check if the rule provides rewrite expressions, and execute them. Then rules of another type were introduced and they detected dependency violations. For example a common feature of many rules is to provide an automatic fix suggestion based on a source code transformation as shown in Figure 3.

Recently architectural critiques were introduced into Pharo, and their special feature is to open a dependency browser and point out the dependency violation. As a result all the tools had to accommodate the new critique with its new feature. Ideally we want to give rule developers the freedom to create new kinds of critiques without having to update all the tools each time. To solve this problem we introduced a concept of *PropertyAction* that has an icon and a description, as well as a "function" that accepts a property and performs the action. A property can have any number of actions, and a tool can list the actions to a programmer and execute them when needed. Figure 4 depicts two possible ways to display actions in the user interface. One of them lists actions as items in a context menu, another uses buttons with icons and a description popup. Whenever a menu item is selected or a button is pressed, the action will be executed.

Figure 5 shows the hierarchy of properties with the actions associated to them. In this case each top-level property introduces one or more actions. The note property opens a text editor to edit the note text; the issue tracker entry opens the issue in a web browser, and the critique can show a detailed description or ban itself, if a developer decides that the report is incorrect. The subclasses of `Critique` inherit actions from their parent and but extend them with the ones specific to their domain. The dependency violation critique can open a dependency browser to provide additional information about dependencies and suggest a solutions. The transformation critique will start a transformation process by displaying the changes that are going to take place and asking the user for approval to execute them.

4.3 Specializing Critiques

It is up to a rule developer to reuse available critiques to create new ones. Based on the demand of custom critiques we integrated some specialized critiques into the base distribution of

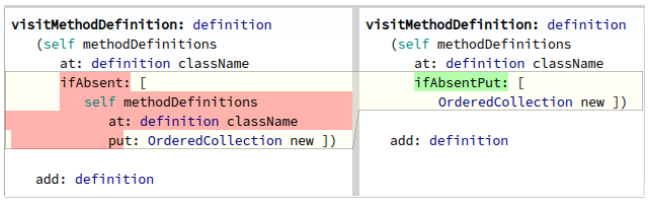

Figure 3. A diff suggesting a fix of a critique.

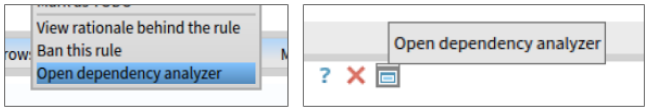

Figure 4. Different user interfaces display actions. On the left the actions are presented as items of a context menu, on the right — as buttons with icons.

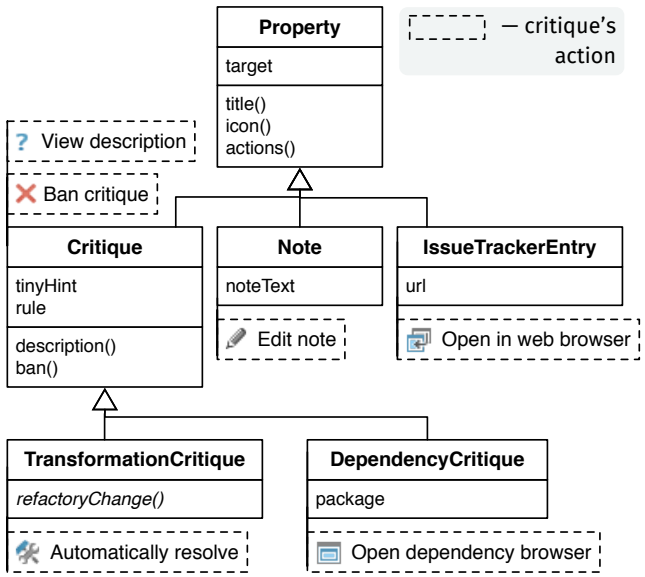

Figure 5. Property hierarchy with the associated actions

the Renraku model. First of all we noticed that many critiques need a possibility to automatically resolve the issue. Thus we introduced TransformationCritique, which knows a transformation that has to be applied and has an action to run the transformation. Then we added more concrete transformation critiques. For example many rules detect a missing method, and a dedicated critique can automatically construct it with a method adding transformation, and communicate which exact method is missing with *tiny hint*. We also noticed that a substantial number of rules detect a wrong order of messages, thus we created a critique that produces an informative title based on the messages and the required order.

5 The Rule

The rule model of Renraku is derived from the existing design of SmallLint — the static analyzer originally available

```
1 rule resetResult.
2 rule checkMethod: aMethod.
3 rule critics includes: aMethod
```

Listing 4. Common way to run programmatically a SmallLint rule. The boolean result represents existence of a violation

```
1 rule check: aMethod
```

Listing 5. Checking a method with a Renraku rule. The result is a collection of critique objects that describe the violations

```
1 check: aClass forCritiquesDo: aCritiqueBlock
2    aClass instVarNames do: [ :varName |
3       varName first isUppercase ifTrue: [
4          aCriticBlock cull: (self critiqueFor:
5             aClass about: varName) ] ]
```

Listing 6. The main checking method of a rule detecting capitalized instance variables

in Smalltalk [8]. Renraku rules share the same properties as SmallLint rules: name, rationale, group, severity. The fundamental difference between SmallLint and Renraku is the ease of use. SmallLint required substantial knowledge about its implementation. To validate code with SmallLint rules one had to use dedicated checkers that had to be reconstructed or reset and queried every time, or run a rule on a source code entity and then query the rule for a result as demonstrated in Listing 4. Various rules had to be queried in a different way which resulted in poor quality reports integrated in tools, as the tool developers did not have time to understand how the rules should be operated. In our case, a rule can be treated as a black box that accepts a target and produces a collection of critiques about that target as demonstrated in Listing 5. Then a tool just has to run a rule and process the obtained critiques.

To achieve the best flexibility and performance we follow a streaming approach where a rule accepts a target to check and a callback function to evaluate for each detected critique. An example of the main checking method taken from the rule that detects capitalized instance variables[5] is shown in Listing 6. The method receives a class to check, and a block to evaluate with each detected critique. Then the method iterates over all the variable names, and in case there is a variable with a first uppercase character, the method creates a critique about this and evaluates[6] the block with the critique. This way the block will be evaluated with every critique representing a single variable violating the capitalization rule. Using the "callback block" approach has a few advantages over returning a collection of critiques. When all the rules

[5] style conventions of Smalltalk define that instance variable names should begin with a lower-case letter

[6] `value:` is the standard Smalltalk method for evaluating a block with one argument. For additional flexibility we use `cull:` in our implementation. Contrary to `value:` it will also evaluate blocks that do not expect any arguments.

are applied to all the methods in Pharo 6 to obtain a single collection of critiques, the streaming approach provides a slight speedup of 10% because it does not create a new collection for every method-rule pair. This approach also allows a tool to run operation-heavy rules in a concurrent process and update the tool UI whenever a critique is detected. Furthermore by using callbacks a developer can stop the rule evaluation on first encountering a critique, if she is interested only in existence of certain critiques and not the detailed report. The rule base class additionally provides convenience methods `check:`, `check:forCritiquesDo:ifNone:` and `check:ifNone:` that return a collection of the detected critiques or accept a block to evaluate if no critiques were detected.

5.1 Specifying a Rule Interest

Another challenge of the rule design is related to distinguishing what type of targets should be checked by a rule. For example one rule can be implemented to check methods, but will break while checking a class. SmallLint solved this by having two empty methods in the root class `checkMethod:` and `checkClass:` and rule runners will only pass a method to the first method and a class to the second one. Then the subclasses only override one of the methods depending on what they want to check. This approach gets more complicated once we have the four checking methods described previously. Additionally, during the evolution of Renraku we had to support rules for checking packages and rules for checking individual AST nodes. For this reason we introduced class-side methods[7] `checksMethod`, `checksClass`, `checksPackage` and `checksNode`. These methods return `false` in the base class and should be overridden to return `true` for rules that are designed to check one of the entity types. A rule may check multiple types of entities, for example a rule that checks if code is correctly packaged may check both methods and classes if they share the same packaging API. This approach allows rule-runners to group all the rules by the type of entity that they are checking and select the appropriate group based on the type of the entity that has to be checked.

During the evolution of Renraku our approach of declaring an interest in a target type worked well. However, when designing the "interest declaration" we envisioned a more complex scenario, when we would also have methods like `checksMetaClass` and `checksMetaClassMethod` that would by default return the value of `checksClass` and `checksMethod` respectively. The concrete rules could override the methods to specify that they want to check only the meta or non-meta entities[8]. We discovered that there is only a small number of rules that distinguish meta and non-meta entities. For rule developers it is easier to validate the meta class details during the checking phase of the rule instead of specifying a special interest with `checksMetaClass` and `checksMetaClassMethod`. As a result we have never implemented special methods for declaring an interest in meta entities. We also envisioned another strategy for declaring entity type interest that may perform better. There could be a single class method which returns an array of types the rule checks. This will not change much for classes, methods, and packages, but can simplify the rules for nodes and introduce a greater flexibility in general. Based on our experience, most of the node-based rules check for the node type in their first operation. For example many rules check something about a message or a variable. Then instead of doing a type check in the rule, developers could specify the type of an AST node they are interested in.

5.2 Specializing Rules

To collect all the rules available in the system we use the same approach used by SmallLint. We simply collect all the subclasses of the abstract rule class and then select ones that check an appropriate target type. Additionally if a certain rule wants to declare an interest in a target for all its subclasses but should not perform validation itself, it can override the *isVisible* to return false for its class and exclude itself from the rules that are used to check code. By dynamically querying the subclasses we can easily add new rules to the existing arsenal if they are packaged with frameworks and libraries that a project uses.

Most of the rules subclass directly the base rule. We also introduced a few custom rules to automate repetitive tasks. One of them is an invocation order rule used to detect whether a certain message is preceded of followed by another one. All such rules traverse an AST and analyze the control flow to detect violations. We generalized the analysis into a common abstract rule and require the concrete subclasses to define only the message pair and the intended invocation order. Another large group of dedicated rules is specialized to check AST nodes. Node-based rules automatically declare interest in AST nodes and override the helper method for constructing source anchors to use the source interval provided by AST nodes. The node-checking rules include a large group of rules that work based on a pattern matching syntax. To create such a rule a developer specifies a source code pattern that should be matched and a transformation which can be used for auto-fix. SmallLint rules based on pattern code traverse the complete AST of a method and rewrite it at the same time, then they store the rewritten version, that can be used by tools to suggest an auto-fix. The Renraku alternative checks a single node and stores the replacement node which is used to apply changes by an auto-fix critique action. This not only allows developers to check a single node, but may speedup code validation by 40%, as an AST does not have to be traversed repeatedly for

[7] In Smalltalk classes are modeled as objects *i.e.,* they have methods too. Class-side methods work similarly to static methods or other programming languages, but can be inherited and overridden.

[8] since classes are objects too, they are instances of meta classes. Meta classes define the class-side variables and methods.

each rule, but can be traversed only once while applying all the pattern code rules to every node.

6 The Target

The target has the least responsibilities to fulfill. According to Renraku any object can be a target. A rule may check a target and produce a critique about it. Targets play an important role of providing a simple API to access critiques. For example all source-code related entities implement a `critiques` method that returns all critiques about this entity by all the active rules in the system. Such a method allows a tool developer to quickly obtain all the critiques about a code entity currently used in a tool. A simplified implementation of such a method is presented in Listing 7. The method is implemented in `Behavior`, which is a common superclass for classes and meta classes thus it knows that it has to check itself with the rules for classes. The method also includes pragma <`eProperty`> because critiques are just one type of property that can exist for this object. For this reason tools are encouraged to actually use another dedicated method `externalProperties` that collects the results from all the methods annotated with <`eProperty`> and aggregates them. The `externalProperties` method is implemented in the root of class hierarchy and thus any object can be asked for its external properties. Then analysis developers may add a method[9] with the <`eProperty`> pragma to a certain class, to make it return their properties together with the others.

```
1  Behavior>>critiques
2     <eProperty>
3     | rules critiques |
4     rules := ReRuleManager uniqueInstance classRules.
5     critiques := OrderedCollection new.
6
7     rules do: [ :rule |
8        rule check: self forCritiquesDo: [ :crit |
9           critiques add: crit ] ]
10    ^ critiques
```

Listing 7. An implementation of a critiques method

7 Compatibility with SmallLint

SmallLint was the static analysis system of Smalltalk for many years before the creation of Renraku. As the result, many rules and tools follow the SmallLint model. To ensure a good migration from SmallLint to Renraku we maintained a healthy level of interoperatibility between the two models. SmallLint rules can be turned into renraku rules with a help of several extension methods in the root class, while a Renraku rule can be turned into a SmallLint rule with the help of a wrapper. The main difference between them is in the checking itself and in the richness of a report. As discussed in Section 5, Renraku rule accepts a target to check and returns a collection of critiques about it. A SmallLint rule has an internal environment where it stores the entities that violate it. When a SmallLint rule checks a code entity and detects a violation it stores the entity in the environment. Then the environment has to be queried for the inclusion of the code entity. Listing 8 demonstrates a Renraku checking method added to an existing SmallLint rule. First of all the method resets the rule's environment which removes all the previously detected violations. Then depending on whether the rule checks classes or methods the corresponding checking message will be sent with the entity as a parameter. In case a violation is detected, the resulting environment will not be empty and thus the method has to produce a critique. To declare an interest in a class or a method we can rely on the rule implementing the corresponding method (Listing 9). The rest of rule properties such as *name*, *rationale*, *severity*, *group* have the same API for both SmallLint and Renraku.

We started the migration by implementing Renraku functionality of the core SmallLint rule and then transforming the available rules one by one. The migration is going to take a long time as there are some external frameworks with SmallLint rules and we have no way to ensure that they have migrated all their rules. The migration could be automated to some extent, but each rule is unique and may store date in different formats, require resets, *etc.* Thus we prefer to have Renraku functionality on top of the existing API and do a manual rule conversion, as Renraku rules may have a better ways of implementation.

```
1  check: anEntity forCritiquesDo: aCritiqueBlock
2     self resetResult.
3     self checkClass:  anEntity.
4     self checkMethod: anEntity.
5     self result isEmpty ifFalse: [ aCriticBlock cull:
6        (self critiqueFor: anEntity) ]
```

Listing 8. Renraku checking based on SmallLint functionality

```
1  checksMethod
2     ^ self theNonMetaClass
3           includesSelector: #checkMethod:
```

Listing 9. Renraku type interest based on SmallLint implementation

Compatibility of Renraku with SmallLint is also important because while someone may decide to convert rules to the Renraku model, certain tools (as Pharo CI server) may still expect SmallLint rules. Because SmallLint is expected to preserve a certain state, we created a wrapper that used a Renraku rule to do the checking while pretending to be a generic SmallLint rule. Because Renraku is explicit about what it checks, the wrapper rule can easily select an appropriate environment, or check a code entity as shown in Listing 10. The challenge arises when a tool asks the rule's class for a `uniqueIdentifierName`, and the wrapper rule is a single class which instances act as diverse rules based on the rule that they wrap. Thus the wrapper rule

[9] Smalltalk allows developers to add methods to classes of other packages.

```
RBRenrakuWrapperLintRule class>>new: aRule
   | annotatedClass |

   annotatedClass := self newAnonymousSubclass.
   annotatedClass class compile:
      'uniqueIdentifierName ^ ',
      aRule class uniqueIdentifierName
         surroundedBySingleQuotes.

   ^ annotatedClass basicNew
      initialize: aRule;
      yourself
```

Listing 11. SmallLint wrapper instantiation

class cannot rely on Renraku rule classes to return a correct `uniqueIdentifierName`. For this reason upon a new wrapper instance creation we also create an anonymous subclass that overrides `uniqueIdentifierName` to return the value provided by the class of the Renraku rule (Listing 11).

```
RBRenrakuWrapperLintRule>>checkClass: aClass
   renrakuRule class checksClass ifFalse: [ ^ self ].
   renrakuRule check: aClass
               forCritiquesDo: [ :crit |
                  result addClass: aClass.
                  ^ self ]
```

Listing 10. SmallLint wrapper class check implementation

8 Creating Rules

In this section we are demonstrating the common workflow to create Renraku rules. To be realistic we are going to look at an issue periodically encountered by Pharo developers. Pharo Catalog[10] is a tool for browsing and quickly installing various projects into Pharo from a dedicated repository. To add a project to Pharo catalog it is not enough to commit a configuration Class to a special repository, but one also must ensure that the configuration has project specific methods. These methods are `catalogDescription`, `catalogContactInfo`, `catalogKeywords` and they provide meta information about the project to be displayed in the catalog. Sometimes developers forget to define these methods and they cannot understand why their projects do not appear in the catalog.

We are going to develop a `ReCatalogRule` which will check if a catalog project configuration defines the required methods. The class will subclass the base `ReAbstractRule` class and override the `checksClass` class-side method to return true. Then we should also comment the class with the rule's rationale, and override the `name`, `severity` and `group` methods to specify important method properties. For this particular case we will also have a helper method `requiredMethods` that returns an array with the selectors of the required methods. The most important part part of the rule is the checking method which is presented in Listing 12. On the lines 2 and 3 we check if the class is a configuration and if it is versioned in the catalog repository to guard ourselves against creating critiques about non-catalog classes. Then we check if there are the required methods on the class-side, and for each missing method we create a critique. At this point the rule has a basic desired functionality. The `critiqueFor:` method that we use creates basic critiques by default, which will report that a class is missing required methods but will not provide information which method is missing. For this reason we have a missing method critique that can be created by implementing a helper method presented in Listing 13. Then this helper method can be used on the line 8 of Listing 12 to produce the critiques that will exactly specify the missing method and offer to create a stub of it. For more complicated rules a developer may want to create a custom critique which can be implemented iteratively once the rule already has a working check method.

```
AbstractRule>>check: aClass
               forCritiquesDo: aCritiqueBlock
   (self testIsConfiguration: aClass)
      ifFalse: [ ^ self ].
   (self testIsInCatalogRepo: aClass)
      ifFalse: [ ^ self ].

   self requiredMethods do: [ :sel |
      (aClass theMetaClass includesSelector: sel)
         ifFalse: [ aCritiqueBlock cull: (
            self critiqueFor: aClass) ] ]
```

Listing 12. The catalog rule checking method

```
AbstractRule>>critiqueFor: aClass missing: aSelector
   ^ ReMissingMethodCritique
      for: aClass
      by: self
      class: aClass theMetaClass
      selector: aSelector)
      beShouldBeImplemented
```

Listing 13. Missing method critique creation for the catalog rule

9 Creating Tools

As mentioned before, the convenient API to obtain critiques greatly simplifies the adoption of static analysis in tools. Designing and building a user interface is a non-trivial task that requires a substantial amount of time and various software components. To simplify the explanation we are going to exemplify the usage of critiques by using them in a software visualization. A standard demonstration of the Roassal [2] visualization framework often includes a script for building a polymetric visualization of a class hierarchy. The visualization depicts classes as rectangles with their width mapped to the number of attributes, height — number of methods and brightness — number of lines of code. The rectangles are connected with edges that represent inheritance between classes and are laid out to form the inheritance tree. Source

[10] http://catalog.pharo.org

code for the visualization together with some features added by us can be seen in Listing 14.

```
1  b := RTMondrian new.
2  b shape box
3     height: #numberOfMethods;
4     width: #numberOfVariables.
5
6  b interaction popupView: [ :group :el |
7     group add: (RTLabel elementOn: el model name).
8     group addAll: (
9        el model critiques collect: [ :crit |
10          crit icon asRTElement ]).
11    RTHorizontalLineLayout on: group ].
12
13 b nodes: RBProgramNode withAllSubclasses.
14 b edges connectFrom: #superclass.
15 b layout tree.
16
17 b normalizer normalizeColor: [ :class |
18    class critiques size ].
```

Listing 14. Roassal script to build a polymetric view for a class hierarchy

We introduced two features into this visualization. First of all, instead of mapping the color of the rectangles to the number of lines of code, we mapped it to the number of critiques. To do this we used a color normalizer on lines 17-18 and specified that the normalization has to be based on the number of critiques[11] of each class. We also updated the popup that appears when a user hovers over an element. Now additionally to showing the name of the class, the popup also contains icons for critiques and their severity. To do this we collect all the icons of the critiques and convert them into Roassal elements, then we add all the resulting elements into the popup group on the lines 8-10. The resulting visualization can be seen on the left-hand side of Figure 6. By using the number of critiques to highlight the classes in red, we can easily draw attention to classes with a high number of critiques. Additionally a user may hover over a class to see its name, the exact number of critiques and their severity. The right-hand side of the figure displays an inspector on the selected object, which is in our case a class that was clicked in the visualization. This is the default behavior, as well as the critiques tab that displays the list of detected critiques. In a tool a developer may implement a similar functionality by obtaining critiques from the object and rendering their icons and descriptions, implementing interactions with them, *etc.* Our main goal is to show that using the static analysis information in tools can be easy, obtaining the number of critiques with only two messages. The tool author can inspect the properties of critiques and use them to provide even more information with a still low implementation cost.

[11] In this example we use `critiques` to avoid complication that comes from the concept of external properties. In reality most of the tools including QualityAssistant use `externalProperties` to include also information of other property engines.

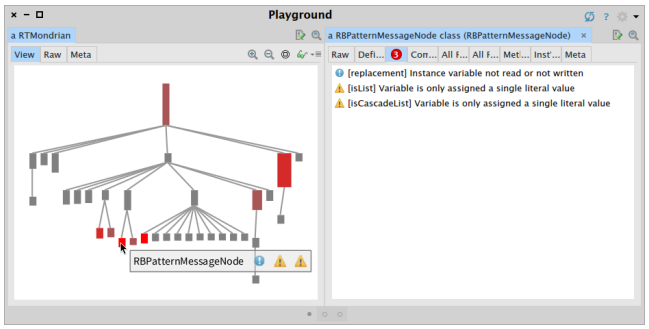

Figure 6. Roassal class hierarchy visualization enhanced with code critiques

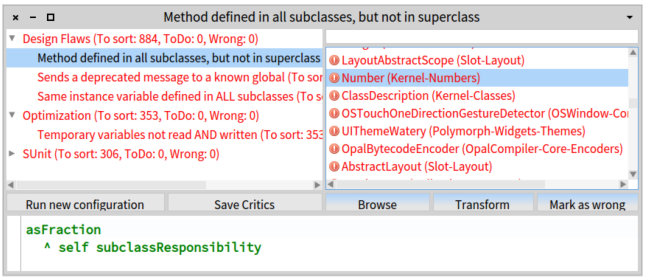

Figure 7. Critique browser suggests a solution to a critique.

10 Notable Users

Renraku is mainly used in QualityAssistant and CriticBrowser which are present in Pharo. Inline critiques of Renraku are also present in the message browser and in the debugger.[12] Renraku was also used in ViDI — the visual design inspector augmented with code quality information [14], but the tool is not being developed for the last two years anymore.

Originally CriticBrowser used SmallLint and the move to Renraku brought a few benefits. Figure 7 shows the CritiqueBrowser suggesting a fix to a missing method critique. SmallLint could not suggest such fix because it could do only method transformations and not a more complicated refactoring. Thus in the original CritiqueBrowser a user would see only a message *"Method defined in all subclasses, but not in superclass"* and the definition of the class where the method is missing without any suggestion which exact method is missing. This example demonstrates how advanced critiques improve all the tools that use Renraku.

Renraku is also used by the Calypso[13] code browser to create a dynamic critiques group that displays critiques for all the methods in a class as can be seen in Figure 8. This use case is very important for Renraku, as the Calypso developer implemented this functionality himself while being an expert in Calypso and not knowing about the rule present in the system, but just relying on the Renraku API.

[12] Debugger critiques are available as a separate loadable plugin.
[13] https://github.com/dionisiydk/Calypso

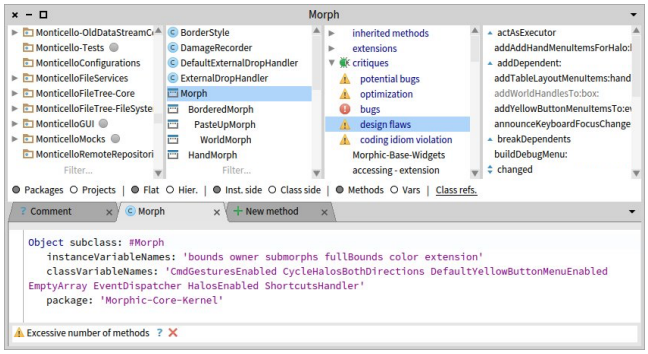

Figure 8. The Calypso browser with a critique method group.

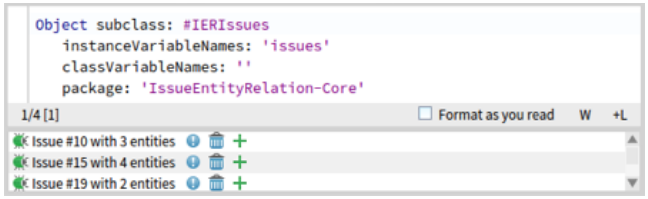

Figure 9. Issue tracker entries displayed in QualityAssistant

Figure 9 depicts a prototype of displaying an issue tracker information related to a code entity in QualityAssistant. While this software was not released to the public, the feasibility of this prototype is an important use case for Renraku. The author of the issue tracker linking engine did not have to learn how to extend a code browser with a plugin, but simply had to return Renraku properties by his engine. Then the tool would automatically pick up and display the properties as they follow the Renraku model.

11 Conclusion

In this paper we presented Renraku — an extensible static analysis model designed to conveniently connect automated software analysis and development tools. The implementation of Renraku was shaped by the requirements that we encountered during our studies. There are prototypes built to demonstrate the flexibility of the framework in combination with various tools. There are also prototypes demonstrating non-rule-based critiques (also known as external properties) and their compatibility with the existing tools. Nonetheless the only setup tested by a substantial amount of real developers during a reasonable amount of time, was the live static analysis feedback performed by QualityAssistant. The model works well and we did not encounter any serious issues during its evolution and operation, and in our previous work we investigated its impact [10, 13].

We believe that Renraku still has a long way to go and many challenges to face. While the concept of a single static analysis model worked for several diverse prototypes, it may include shortcomings that can be revealed only when tested by a reasonable number of real users. We will not know much about its usefulness until more analysis engines start to provide their feedback using the model, and more tools will embed the Renraku external properties into their interface.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Analysis" (SNSF project No. 200020-162352, Jan 1, 2016 - Dec. 30, 2018).

References

[1] Nathaniel Ayewah and William Pugh. 2008. A Report on a Survey and Study of Static Analysis Users. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems (DEFECTS '08)*. ACM, New York, NY, USA, 1–5. DOI:http://dx.doi.org/10.1145/1390817.1390819

[2] A. Bergel. 2016. *Agile Visualization*. LULU Press. https://books.google.ch/books?id=lEk7vgAACAAJ

[3] Alexandre Bergel and Vanessa Pe na. 2012. Increasing test coverage with Hapao. *Science of Computer Programming* 79, 1 (2012), 86–100. DOI:http://dx.doi.org/10.1016/j.scico.2012.04.006

[4] Tim Buckers, Clinton Cao, Michiel Doesburg, Boning Gong, Sunwei Wang, Moritz Beller, and Andy Zaidman. 2017. UAV: Warnings from Multiple Automated Static Analysis Tools at a Glance. In *2017 IEEE 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 472–476.

[5] Tommaso dal Sasso, Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. Blended, Not Stirred: Multi-concern Visualization of Large Software Systems. In *Proceedings of VISSOFT 2015 (3rd IEEE Working Conference on Software Visualization)*. 106–115. DOI:http://dx.doi.org/10.1109/VISSOFT.2015.7332420

[6] Stéphane Ducasse, Dmitri Zagidulin, Nicolai Hess, and Dimitris Chloupis. 2017. *Pharo by Example 5.0*. Square Bracket Associates. http://files.pharo.org/books/updated-pharo-by-example/

[7] Angela Lozano, Kim Mens, and Andy Kellens. 2015. Usage contracts: Offering immediate feedback on violations of structural source-code regularities. *Science of Computer Programming* 105 (2015), 73 – 91. DOI:http://dx.doi.org/10.1016/j.scico.2015.01.004

[8] Don Roberts, John Brant, Ralph E. Johnson, and Bill Opdyke. 1996. An Automated Refactoring Tool. In *Proceedings of ICAST '96, Chicago, IL*.

[9] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 598–608. http://dl.acm.org/citation.cfm?id=2818754.2818828

[10] Yuriy Tymchuk. 2015. What if Clippy Would Criticize Your Code?. In *BENEVOL'15: Proceedings of the 14th edition of the Belgian-Netherlands software evoLution seminar*. http://yuriy.tymch.uk/papers/benevol15.pdf

[11] Yuriy Tymchuk. 2017. QualityAssistant v3.3.1. (June 2017). DOI:http://dx.doi.org/10.5281/zenodo.809410

[12] Yuriy Tymchuk. 2017. Renraku v0.15.2. (May 2017). DOI:http://dx.doi.org/10.5281/zenodo.800676

[13] Yuriy Tymchuk, Mohammad Ghafari, and Oscar Nierstrasz. 2016. When QualityAssistant Meets Pharo: Enforced Code Critiques Motivate More Valuable Rules. In *IWST '16: Proceedings of International Workshop on Smalltalk Technologies*. 5:1–5:6. DOI:http://dx.doi.org/10.1145/2991041.2991046

[14] Yuriy Tymchuk, Andrea Mocci, and Michele Lanza. 2014. Collaboration in open-source projects: myth or reality?. In *MSR'14: Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 304–307. DOI:http://dx.doi.org/10.1145/2597073.2597093