

# Architecture Conformance Checking in Dynamically Typed Languages

Sergio Miranda<sup>a</sup>   Elder Rodrigues Jr<sup>b</sup>   Marco Tulio Valente<sup>a</sup>

Ricardo Terra<sup>b</sup>

a. Department of Computer Science, Federal University of Minas Gerais

b. Department of Computer Science, Federal University of Lavras

**Abstract** Architectural erosion is a recurrent problem faced by software architects, which might be even more severe in systems implemented in dynamically typed languages. The reasons are twofold: (i) some features provided by such languages make developers more propitious to break the planned architecture (e.g., dynamic invocations and bindings), and (ii) the developers' community lacks tool support for monitoring the implemented architecture. To address these shortcomings, this paper presents an architectural conformance and visualization approach based on static code analysis techniques and on a lightweight type propagation heuristic. The central idea is to provide the developers' community with means to control the architectural erosion process by reporting architectural violations and visualizing them in high-level architectural models, such as reflexion models and DSMs. This paper also describes a tool—called ArchRuby—that implements our approach. We evaluate our solution in three real-world systems identifying 48 architectural violations of which the developers had no prior knowledge. We also measure the effectiveness of our type propagation heuristic reporting that (i) the number of analyzed types raises 5% on the average and (ii) certain violations are only detected due to our heuristic.

**Keywords** Architecture conformance checking; high-level architectural models; dynamically typed languages.

## 1 Introduction

The planned architecture of a system comprises a set of standards and best practices that enable its evolution [Par94]. However, as the project evolves—due to lack of knowledge, short deadlines, etc.—these patterns tend to deteriorate and hence nullifying the benefits provided by an architectural design, such as maintainability, scalability, portability, etc. [PTD<sup>+</sup>10, MNS95]. This phenomenon is known as software

architecture erosion and it is considered a challenging research problem in the software architecture area [KMHM08, TVCB15, SRK<sup>+</sup>09, Bor11, Bos04]. This erosion process might be even more severe in systems implemented in dynamic languages for two main reasons: (i) some features provided by these languages (e.g., dynamic invocations, dynamic bindings, eval, etc.) make the developers more likely to break the planned architecture, and (ii) these languages suffer from the lack of architecture and design tools.

This article is centered on the assumption that systems implemented in dynamic languages should also benefit from architectural conformance techniques and tools. More important, existing architectural conformance solutions are limited since, as far as we know, none addresses the particularities of dynamically typed languages. Therefore, in a previous conference paper [MVT15b], we were the precursors in proposing an architectural conformance checking approach for systems implemented in dynamic languages, which is based on static code analysis techniques and on a lightweight type propagation heuristic. Although our focus was on the architecture conformance checking, we also provided high-level architectural models to better visualize the identified violations. Furthermore, we showed that it is possible to monitor the architecture of these systems using our proposed approach, which is non-invasive and hence does not modify the source code or impact on the performance. As a practical contribution, we implemented a tool for Ruby that supports a simple and objective way to detect architectural violations and to visualize them using two high-level architectural models, namely Reflexion Models and Dependency Structure Matrices (DSMs).

In the presented paper, we extend our work in the following directions: (a) by including more details and one more system in the evaluation of the proposed approach. As a result, we could detect 48 violations in three real-world systems of which the developers had no prior knowledge; (b) by evaluating the effectiveness of our lightweight type propagation heuristic in the three previously evaluated real-world systems, besides complementarily in 28 open-source systems. As main findings, the number of inspected types increases 5% on the average but up to 17% with our heuristic, and some violations are only detected due to this heuristic; (c) by applying **ArchRuby** in itself we illustrate our approach; and (d) by providing more technical details on the **ArchRuby** implementation.

The remainder of this paper is organized as follows. Section 2 provides a definition for central concepts needed to follow our approach, such as Ruby features, architecture conformance checking, and high-level architectural models. Section 3 presents the proposed approach, describing the architectural rules specification, conformance, and visualization processes. Section 4 details the proposed type propagation heuristic. Section 5 presents **ArchRuby**, the tool that implements the proposed solution. Section 6 reports results from applying our solution in three real-world systems. Section 7 measures the effectiveness of the proposed type propagation heuristic in the previous evaluated systems. Finally, Section 8 discusses related work and Section 9 concludes.

## 2 Background

In this section, we discuss background related to our work. Section 2.1 describes and exemplifies Ruby features, and Section 2.2 briefly introduces architectural conformance checking techniques and high-level architectural models.

## 2.1 Ruby

Since our approach focuses on systems implemented in Ruby, an overview of the language features is relevant. Every Ruby program is designed through objects, since the language is purely objected-oriented [Bla09]. Even *true* and *false* are objects, i.e., they are instances of `TrueClass` and `FalseClass`, respectively. To define classes and methods, we use the `class` and `def` keywords, respectively. Ruby has single inheritance, but it is possible to include many modules in one class. Basically, a module is used to implement methods and constants but, unlike a class, it is not possible to instantiate a module.

Furthermore, Ruby is a dynamic language with several powerful abstractions. For example, it is possible to define methods, classes, re-open a class, evaluate a valid Ruby code inside some context, re-define methods and call methods passing strings as arguments. As an example, Figure 1 illustrates the usage of the aforementioned features. In line 1, the code is defining a class `RbClass` and a module `Test`. In line 2, class `RbClass` includes module `Test`, therefore method `salute` is now part of this class. In line 7 we instantiate an object of `RbClass` and call method `say_hi` (line 8). In line 10, we re-open class `RbClass` and define a new method called `say_bye` (lines 11-13). In lines 16-18, we define a method `add` only for the object `o`. Finally, in line 20, we call the method defined by last.

```

1  class RbClass          module Test
2    include Test        def salute
3    def say_hi          puts "Hi"
4      salute            end
5    end                 end
6  end
7  o = RbClass.new
8  o.say_hi
9
10 class RbClass
11   def say_bye
12     puts "bye"
13   end
14 end
15
16 def o.add(x,y)
17   x.send "+", y
18 end
19
20 o.add(5,9)

```

Figure 1 – Ruby source code example

## 2.2 Architectural Conformance and Visualization

Architecture is a crucial artifact that needs to be followed and monitored by the developers during software development. Architectural conformance is the process that checks to which degree the concrete architecture (e.g., the source code implementation) is consistent with the planned one [KMR08]. Architectural conformance can be static

(i.e., without executing the target system) or dynamic (i.e., executing the target system). This section presents architecture conformance checking techniques related to ArchRuby:

**Reflexion Models (RMs):** As proposed by Murphy et al., Reflexion Models compare two models [MNS95]. One representing a high-level model of the system (e.g., specified by the developer) and another representing the low-level model of the system (e.g., produced either by statically analyzing the system source or by collecting information during the system execution). With these inputs the technique computes a software reflexion model that matches the high-level and the low-level models. The reflexion model highlights divergences and absences, regarding these models. Divergences indicate source interactions that are not expected by the planned architecture and absences indicate interactions that are expected but not found. The outcome of the evaluation is usually summarized and documented in a separated report, which is presented as a graph and text. The former connects system modules and reports the detected divergences and absences.

**Dependency Structure Matrices (DSMs):** The concept of DSM was first proposed by Baldwin and Clark to show the importance of modular design in the hardware industry [BC99]. Thereafter, Sullivan et al. claimed that DSM could also be used in software industry [SGCH01]. A DSM is a square matrix where the rows and columns represent the modules of the system. Traditionally, DSM used a “X” to indicate a dependency between two modules. However, Sangal et al. in the LDM tool represent in the cells the number of references between two modules [SJSJ05]. In this tool, it is possible to distinguish the dependencies using design rules, which have two forms: *A can-use B* and *A cannot-use B*, indicating that module *A* can (or cannot) depend on module *B*. DSM has a more scalable output than the output generated by reflexion models based on graphs, since a matrix usually scales better than a graph.

**Constraint languages:** The main objective of constraint languages is to provide a method to specify structural dependencies. DCL (Dependency Constraint Language) is a domain specific language that supports the definition of structural constraints between modules [TV09]. DCL provides constraints to capture divergences and absences. First, to capture divergence architects have to specify **only can**, **can only** or **cannot** rules for specified modules. Last, to capture absence architects specify dependencies that **must** be present in the source code. ArchRuby—the architecture conformance checking technique proposed in this paper—is directly inspired on DCL constraints.

### 3 The Proposed Approach

This paper describes an architectural conformance approach based on static code analysis techniques and on a lightweight type propagation heuristic for systems implemented in dynamically typed languages. Additionally, we also provide two high-level architectural models to better visualize the architectural violations. The central goal is to provide developers with means to control the architectural erosion process by reporting architectural violations (conformance) and by providing the high-level architectural models to better visualize the identified violations (visualization).

Figure 2 provides an overview on the proposed approach. Our solution receives as input the architectural rules (*in1*) and the source code of the target system (*in2*). After parsing the architectural rules file (*t1*) and the source code (*t2*), it triggers the architectural conformance process (*t3*) in order to detect design decisions that do not respect the intended architecture. As result, our solution outputs a textual report (*out1*), which details the detected violations (source code location, violated rule, etc.), and two high-level architectural models to better visualize the identified violations (out2). In these models, we differentiate the dependencies—edges in reflexion models and cells in DSMs—that represent violations (refer to Section 2.2).

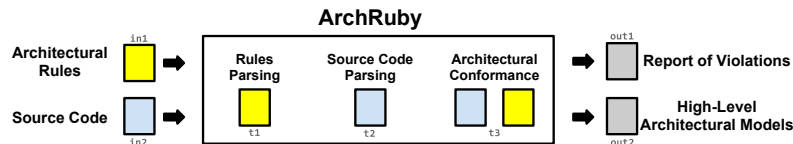


Figure 2 – The proposed approach

This section is organized as follows. Section 3.1 presents the running system. Section 3.2 details the specification of architectural rules and Section 3.3 describes the architectural conformance process. Finally, Section 3.4 describes the high-level architectural models our approach relies on to better visualize the identified violations.

### 3.1 Running Example

We rely on the architecture of ArchRuby<sup>1</sup> itself and its implementation to illustrate the architectural conformance and visualization processes provided by our approach. The tool was implemented in Ruby and relies on five Gems:<sup>2</sup> `RubyParser` to parse the source code, `SexpProcessor` to perform tree traversals, `Yaml` to parse the architectural rules specification file, `GraphViz` to produce the reflexion model, and `IMGKit` to produce the DSM. Figure 3 shows the diagram of the core classes of the system. A more detailed description on the ArchRuby implementation can be found in Section 5.

### 3.2 Architectural Rules Specification

Architectural rules are specified in a domain-specific language in YAML format, widely used in the Ruby ecosystem. Thereupon, even non-experienced developers can easily define rules. Specifically, each module of the system under evaluation must be formalized as follows:<sup>3</sup>

```

1 <module_id>:
2   (files | gems): '<pattern_desc> {,<pattern_desc>}'
3   [(allowed | forbidden): '<module_id> {,<module_id>}']
4   [(required): '<module_id> {,<module_id>}']

```

where *<module\_id>* is the name of the module (line 1). Modules can be composed by files (*files*) or Gems (*gems*) that must be defined by at least one *<pattern\_desc>*,

<sup>1</sup>The source code is publicly available at <http://github.com/sergiotp/archruby>.

<sup>2</sup>Gem represents a reusable package or application written in Ruby language.

<sup>3</sup>Formalization based on the Extended Backus-Naur Form (EBNF).

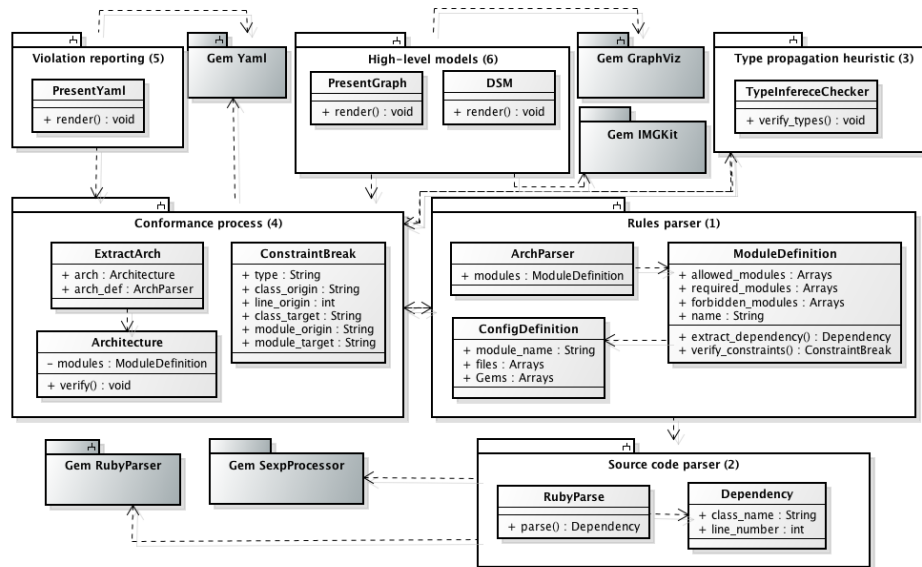


Figure 3 – ArchRuby architecture

delimited by commas (line 2). It is not possible to (i) combine files and Gems in the same module definition, and (ii) define constraints to module composed strictly by Gems, since they are external libraries that are not part of the target system being analyzed. When specifying files, the pattern matching is based on *shell glob*<sup>4</sup> (a default Ruby file library) to map multiple files at once using wildcards, e.g., \* and \*\*.

To detect divergences—dependencies that exist in the source code but are not prescribed by the planned architecture [PTD<sup>+</sup>10]—for each module we define the ones that it is allowed to depend (**allowed**) or not (**forbidden**), which are defined by at least one `<module_id>`, delimited by commas (line 3). Here, we consider as a dependency from a type *A* to a type *B* when (i) *A* accesses a field of type *B*, (ii) *A* invokes a method of type *B*, (iii) *A* instantiates an object of type *B*, (iv) *A* declares a variable or formal parameter of type *B*, (v) *A* raises an exception of type *B*, and (vi) *A* inherits from, extends, or includes *B*.<sup>5</sup> Likewise, to detect absences—dependencies that do not exist in the source code but are required by the planned architecture [PTD<sup>+</sup>10]—for each module we define the ones that it must depend (**required**), which are defined as aforementioned (line 4). It is worth noting that a definition for a particular module can combine **required** with **allowed** or **forbidden**. However, it cannot have **allowed** and **forbidden** in a same module definition. When a module does not define clauses **allowed** and **forbidden**, our language considers that such module is allowed to depend on any module.

In order to illustrate an YAML definition, Figure 4 presents the definition of

<sup>4</sup>A detailed explanation of shell glob in Ruby (specifically, class `Dir`) can be found at: <http://ruby-doc.org/core-2.2.0/Dir.html#method-c-glob>

<sup>5</sup>The code of a lambda is verified only in the method where it is defined, not in its call sites. For instance, assume that a method `return_lambda` in module  $M_3$  returns a lambda  $f$ . Assume also that a module  $M_2$  defines a method `search_lambda` that calls  $M_3::return_lambda$ . Assume, lastly, that a method in module  $M_1$  calls  $M_2::search_lambda$ . In such scenario, (i) only module  $M_3$  depends on the types lambda  $f$  establishes dependency with, (ii) module  $M_1$  depends only on module  $M_2$ , and (iii) module  $M_2$  depends only on module  $M_3$ .

architectural rules to the ArchRuby tool. For example, module `module_definition` (lines 1-3) contains file `module_definition.rb` and `can` depend on classes from module `config_definition`, `ruby_parser`, `dependency`, `constraint_break`, and `file_extractor`. On the other hand, module `multiple_constraints_validator` (lines 5-7) contains file `archruby.rb` and `cannot` depend on classes from module `architecture`. Moreover, `shell glob` allows to use `*` to reference all files in the directory and `**` to reference directories in a recursive manner. For example, module `presenters` (line 13) is composed by all `rb` files listed in directories inside `presenters`. It is worth noting that we do not define architectural rules for modules strictly composed by Gems (e.g., `parser_ruby`, `sexp_processor`, `yaml_parser`, and `graphviz`) because they are not internal components of the target system. Nevertheless, Gems must be defined by their namespace (main module). For example, module `parser_ruby` is composed by Gem `ruby_parser` whose namespace is `RubyParser` (lines 41-42).

```

1 | module_definition:
2 |   files: 'lib/archruby/architecture/module_definition.rb'
3 |   allowed: 'config_definition, ruby_parser, dependency, constraint_break, file_extractor'
4 |
5 | multiple_constraints_validator:
6 |   files: 'lib/archruby.rb'
7 |   forbidden: 'architecture'
8 |
9 | architecture_parser:
10 |  files: 'lib/archruby/architecture/parser.rb'
11 |  allowed: 'config_definition, module_definition, type_propagation, yaml_parser'
12 |
13 | presenters:
14 |  files: 'lib/archruby/presenters/**/*.*.rb'
15 |  allowed: 'architecture, graphviz, imgkit'
16 |
17 | ruby_parser:
18 |  files: 'lib/archruby/ruby/parser.rb'
19 |  allowed: 'dependency'
20 |  required: 'parser_ruby, sexp_processor'
21 |
22 | config_definition:
23 |  files: 'lib/archruby/architecture/config_definition.rb'
24 |
25 | architecture:
26 |  files: 'lib/archruby/architecture/architecture.rb'
27 |  forbidden: 'type_propagation'
28 |
29 | constraint_break:
30 |  files: 'lib/archruby/architecture/constraint_break.rb'
31 |
32 | dependency:
33 |  files: 'lib/archruby/architecture/dependency.rb'
34 |
35 | type_propagation:
36 |  files: 'lib/archruby/architecture/type_propagation_checker.rb'
37 |
38 | file_extractor:
39 |  files: 'lib/archruby/architecture/file_content.rb'
40 |
41 | parser_ruby:
42 |  gems: 'RubyParser'
43 |
44 | sexp_processor:
45 |  gems: 'SexpInterpreter'
46 |
47 | yaml_parser:
48 |  gems: 'YAML'
49 |
50 | graphviz:
51 |  gems: 'GraphViz'
52 |
53 | imgkit:
54 |  gems: 'IMGKit'

```

Figure 4 – ArchRuby architectural specification file

### 3.3 Architectural Conformance

The architectural conformance process is performed from the architectural rules specification and the source code of the target system. This process (i) extracts the modules and rules from the architectural rules specification file; (ii) extracts the dependency graph of the entire system; (iii) includes type information in the dependency graph using a type propagation heuristic (described in Section 4); and (iv) checks whether the dependencies obtained in steps *ii* and *iii* respect the rules defined in step *i*.

The conformance process outputs a file reporting the detected architectural violations (divergences and absences). For example, consider the rules defined for ArchRuby (Figure 4). In such specification, module `module_definition` is not explicitly allowed to depend on module `type_propagation` (line 3). However, assume that a class from `module_definition` accesses a class from `type_propagation`. Such dependency represents a violation and would be reported to developers in the textual output file as illustrated in Figure 5.<sup>6</sup> For each detected violation, the report indicates the violation type (line 1), information from the origin class (lines 2–4) and from the target class (lines 5–6), and the violated rule (line 7). Besides the textual report file, ArchRuby also provides two graphical report files in order to provide complementary ways to visualize the detected violations, as explained in Section 3.4.

```

1 divergence:
2   origin_module: module_definition
3   origin_class: Archruby::Architecture::ModuleDefinition
4   origin_line: 29
5   target_module: type_propagation
6   target_class: Archruby::Architecture::TypePropagationChecker
7   constraint: module 'module_definition' cannot depend on module 'type_propagation'

```

Figure 5 – Textual report of an architectural violation

### 3.4 Architectural Visualization

Although we focus on architecture conformance checking process, we complement our textual report of violations by providing two high-level architectural models to better visualize the identified violations: (i) Reflexion Model in a subtle adaptation of the one originally proposed by Murphy et al. [MNS95] and (ii) Dependency Structure Matrix (DSM) in a subtle adaptation of the one proposed by Sangal et al. [SJSJ05].

#### 3.4.1 Reflexion Model

The reflexion model is a directed dependency graph whose vertices represent the modules defined in the architectural rules specification and edges represent dependencies established between the modules, which are differentiated when refer to architectural violations (refer to Section 2.2).

Figure 6 illustrates the reflexion model of ArchRuby.<sup>7</sup> The light gray rectangles represent internal modules (e.g., `module_definition`) and the gray trapezes represent external modules (e.g., `parser_ruby`). The edges are shown as follows (assume an edge from *A* to *B*):

<sup>6</sup>The report is also in YAML format to facilitate reuse.

<sup>7</sup>For a better visualization, all high-level architectural models—Reflexion Models and DSMs—are publicly available at: <http://aserg.labsoft.dcc.ufmg.br/archruby/jot2015>



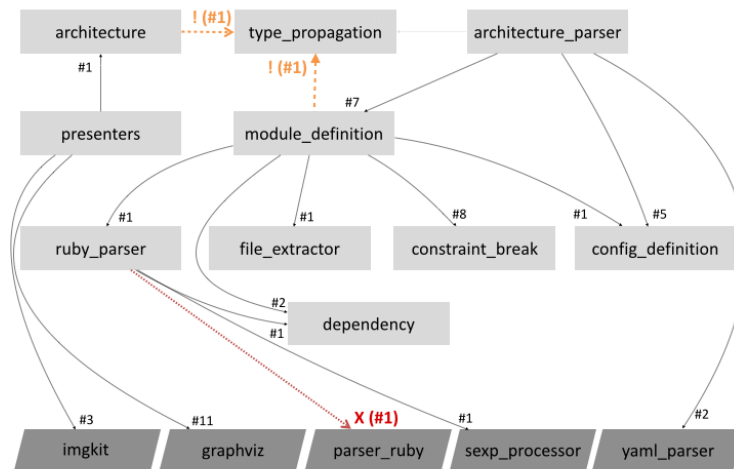


Figure 6 – Reflexion model automatically computed by ArchRuby





- ( $\rightarrow$ ) Black edge: indicates an *allowed* dependency from module  $A$  to  $B$ . For instance, `ruby_parser` establishes one ( $\#1$ ) dependency with module `dependency` (see line 19, Figure 4).
- ( $\dashrightarrow$ ) Dashed orange edge with an “!” mark: indicates a *divergence*, i.e., there is a class from module  $A$  depending on module  $B$ , even though it is (i) *forbidden* or (ii) not explicitly *allowed*. For example, `architecture` depends on module `type_propagation`, but it is forbidden (case  $i$ ; see line 27, Figure 4). As another example, `module_definition` depends on module `type_propagation`, but it is not explicitly allowed (case  $ii$ ; see line 3, Figure 4).
- ( $\cdot \times$ ) Dotted red edge with an “X” mark: indicates an *absence*, i.e., there is no class from module  $A$  depending on module  $B$ , even though it is *required*. For instance, a class from `ruby_parser` does not depend on `parser_ruby` (see line 20, Figure 4).
- ( $\rightarrow$ ) Gray edge: indicates a *warning*, i.e., there is no class from module  $A$  depending on module  $B$ , even though it is prescribed as *allowed*. For instance, we defined that `architecture_parser` is allowed to depend on module `type_propagation` (see line 11, Figure 4), but there is no dependency from the former to the latter.

### 3.4.2 Dependency Structure Matrix

Reflexion models have a well-known scalability problem since it is a graph-based model. As the number of modules and dependencies grows, the model becomes unreadable. In this sense, ArchRuby also provides a high-level architectural model based on DSMs,

which is a weighted square matrix where the rows and columns are numbered and represent the modules of the system, and the cells represent the dependencies between them (refer to Section 2.2).

Figure 7 illustrates the DSM of ArchRuby. The cells represent the number of references between two modules. The cells are shown as follows:

- (  ) Gray cell: indicates a *allowed* dependency. For instance, the number 7 in row 1 and column 3 denotes that module `architecture_parser` establishes seven *allowed* dependencies with module `module_definition`.
- (  ) Orange cell: indicates a *divergence*. For example, the number 1 in row 10 and column 7 represents that module `architecture` establishes a *forbidden* dependency with module `parser_ruby`. As another example, the number 1 in row 10 and column 1 represents that module `module_definition` establishes a *forbidden* dependency with module `type_propagation`.
- (  ) Red cell: indicates an *absence*. For instance, the number 1 in row 12 and column 5 represents that module `ruby_parser` does not depend on module `parser_ruby` even though it is required.
- (  ) Question cell: indicates a *warning*. For instance, the symbol “?” in row 10 and column 3 represents that module `architecture_parser` does not establish an *expected* dependency with module `type_propagation`.

Modules	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
module_definition	1		7													
multiple_constraints_validator	2		1													
architecture_parser	3	1														
presenters	4															
ruby_parser	5	1														
config_definition	6	1	5													
architecture	7	1	1													
constraint_break	8	8														
dependency	9	2			1											
type_propagation	10	1	?				1									
file_extractor	11	1														
parser_ruby	12				1											
sexp_processor	13				1											
yaml_parser	14		2													
graphviz	15			11												
imgkit	16			3												

Figure 7 – DSM automatically computed by ArchRuby

## 4 The Proposed Type Propagation Heuristic

In this section, we describe a type propagation heuristic—more specifically, a simplification of the one formalized by Furr et al. [FhDAFH09]—which aims to build a set `TYPES` whose elements are triples `[method, var_name, type]`, where `type` is one of the possible types inferred for a variable or a formal parameter `var_name` defined in method `method`. We build this set based on the following recursive definition:

- i) **Base:** For each direct inference (e.g., instantiation) of a type  $T$  assigned to a variable  $x$  in a method  $f$ , then  $[f, x, T] \in \text{TYPES}$ .
- ii) **Recursive step:** If  $[f, x, T] \in \text{TYPES}$  and there is a call  $g(x)$  in  $f$ , then  $[g, y, T] \in \text{TYPES}$ , where  $y$  is the name of the formal parameter in  $g$ . This step is applied until a fixpoint is reached, i.e., no new triples are added to set  $\text{TYPES}$ .

Figure 8 illustrates the proposed heuristic. When executing the base step of the algorithm, it initializes  $\text{TYPES}$  with  $[A::f, x, \text{Foo}]$ ,  $[A::f, b, B]$ ,  $[A::f, \text{self}, A]$ ,  $[B::g, c, C]$ , and  $[C::h, d, D]$  since they can be directly inferred. On the first application of the recursive step, the triples  $[B::g, x, \text{Foo}]$  and  $[B::g, z, A]$  are included in  $\text{TYPES}$ , since the type of the variables  $x$  and  $\text{self}$  are known in the call of  $g$ . On the second application of the recursive step, the triples  $[C::h, y, \text{Foo}]$  and  $[C::h, y, A]$  are included in  $\text{TYPES}$ , since the type of variables  $x$  and  $z$  are known in the call of  $h$ . On the third application of the recursive step, the triples  $[D::m, k, \text{Foo}]$  and  $[D::m, k, A]$  are included in  $\text{TYPES}$  (where  $k$  is the name of the parameter in  $D::m$ ), since the type of the variable  $y$  is known in the call of  $m$ . The fourth application of the recursive step reaches the fixpoint since no new triple is added to set  $\text{TYPES}$ .

1	<code>class A</code>	2	<code>class B</code>	3	<code>class C</code>
2	<code>  def f</code>	3	<code>  def g(x z)</code>	4	<code>  def h(y)</code>
3	<code>    x = Foo.new</code>	4	<code>    c = C.new</code>	5	<code>    d = D.new</code>
4	<code>    b = B.new</code>	5	<code>    c.h(x)</code>	6	<code>    d.m(y)</code>
5	<code>    b.g(x, self)</code>	6	<code>    c.h(z)</code>	7	<code>  end</code>
6	<code>  end</code>	7	<code>  end</code>		<code>end</code>
7	<code>end</code>		<code>end</code>		

Figure 8 – Piece of code to illustrate the proposed type propagation heuristic

In this example, it is worth noting that the formal parameter  $y$  of method  $C::h$  can be either of type  $A$  or  $\text{Foo}$ . It indicates that: (i) the type propagation mechanism has to consider all potential types of a variable or formal parameter when propagating the type; and (ii) the architectural conformance process has also to consider all potential types ( $A$  and  $\text{Foo}$ , in this scenario) when searching for violations.

## 5 The ArchRuby Tool

`ArchRuby` is a Gem for Ruby that implements our proposed approach [MVT15a]. The tool is executed from the command line. We decided for such UI because, in such way, any organization—regardless of its software environment—can adopt `ArchRuby` in its development process. The following example illustrates a usage scenario:

```
archruby --arch_def_file=/fmot/arch_def.yml --app_root_path=/fmot
```

The executable `archruby` requires as input the path of the architectural rules file (`--arch_def_file`) and the path of the system (`--app_root_path`), and provides as output the architecture violation report (`archruby_report.yml`) and two high-level

architectural models to better visualize the identified violations (`archruby_rm.png` and `archruby_dsm.png`), as previously illustrated in Figure 2.

As also previously illustrated in Figure 3, the ArchRuby implementation follows an architecture divided in the following modules:

1. *Rules parser*: Responsible for extracting and storing the content of the architectural rules file (e.g., `/fmot/arch_def.yml`) in an internal data structure. It also warns the user when he/she specifies invalid constraints, e.g., `allowed` and `forbidden` together. We rely on the standard Ruby `Yaml` Gem to parse the YAML file.
2. *Source code parser*: Responsible for extracting and storing all system dependencies (e.g., from `/fmot`) in an internal data structure. We rely on Gem `ruby_parser` to parse the source code of each class. It produces *s*-expressions, which are data structures in form of tree. Basically, during the tree traversal, this module stores the type of variables and formal parameters, besides the calls involving them.
3. *Type propagation heuristic*: Responsible for inferring types of variables, according to the heuristic previously described in Section 4. It complements the internal data structure obtained by the *Source code parser* module.
4. *Conformance process*: Responsible for verifying whether the implemented architecture (as represented by the source code) follows the planned architecture (as represented by the architectural rules), as previously described in Section 3.3. This module detects the dependencies that do *not* respect the specified architectural rules and stores detailed information regarding them. It relies on the data structures initially built by the *Rules parser* and *Source code parser* modules to detect the dependencies that do not respect the architectural rules. In other words, this module analyzes the internal data structure built in the previous steps to search for potential violations. When a violation is detected, it stores detailed information—namely dependency type, name of the source and target modules, line number, and name of the source and target classes (see Figure 3, class `ConstraintBreak`)—for further reference.
5. *Violation reporting*: Responsible for structuring the detected architectural violations in a YAML file (`archruby_report.yml`).
6. *High-level models*: Responsible for generating the high-level architectural models of the target system as previously described in Section 3.4. It relies on the data structure initially built by the *Source code parser* module and on the set of violations detected in the *Conformance process* module to highlight the identified violations in the generated visualization models. This module relies on Gem `GraphViz` to produce reflexion models as annotated directed dependency graphs and on Gem `IMGKit` to produce DSMs as HTML tables with CSS style.

Although each of the aforementioned modules has a well-defined responsibility, they may contain more than one single class in order to have a greater control over the parts of the system. In such way, it is easier to maintain the existing features and add new ones. Furthermore, we have implemented several unit tests that are automatically

performed during regression testing to ensure that changes do not break the expected behavior of the system. It is worth noting that the dependencies that are not part of the Ruby standard library (e.g., `ruby_parser` and `GraphViz`) are automatically installed when the user installs `ArchRuby`.

## 6 Evaluation of the Proposed Approach

This section evaluates the applicability of our proposed approach in real contexts of software development. We chose three real-world systems—`Dito Social`, `Tim Beta`, and `PLC Attorneys`—to apply our architecture conformance checking process. For each system, we report the results into each step: (i) *architectural rules specification*, (ii) *architectural conformance*, and (iii) *architectural visualization*. More important, a qualitative discussion is conducted for each evaluated system, besides a general discussion to conclude the section.

### 6.1 Target Systems

We evaluate our solution in three real-world systems:<sup>8</sup> `Dito Social`, a social platform provided by an IT company to its final customers; `Tim Beta`, a telecommunication company communication channel with mostly target young groups; and `PLC Attorneys`, a project task management software system used by a law firm. Table 1 reports the main information of the systems.

Table 1 – Target systems

System	LOC	# classes / # gems	Technologies
<code>Dito Social</code>	13,304	142 / 34	Ruby on Rails, Resque, Rspec, RSA, Twitter, Google Plus, Koala, Suspot Rails, Mysql2
<code>Tim Beta</code>	17,817	141 / 50	Ruby On Rails, Resque, Twitter, YoutubeIt, Google Plus, Instagram, Devise, Foursquare2
<code>PLC Attorneys</code>	2,034	52/35	Ruby on Rails, Devise, CanCanCan, PaperClip, Mysql2, Select2Rails, CoffeeRails

### 6.2 Methodology

For each subject system with the support of its chief architect who designed the evaluated architecture, we performed the following major steps:

- (i) *Architectural rules specification*: The software architect defines the planned architecture of the system, soon after be instructed on how to specify modules and rules using our architectural description language (Section 3.2). To ensure the correct understanding by the architects, we ask them to practice the specification in an illustrative project. During the practice, they must specify a few rules and they can ask for clarifications. By concluding the practice, we argue that the architects are fully qualified to specify the architectural rules.
- (ii) *Architectural conformance*: After a brief tutorial about our tool—its inputs and outputs—the software architect executes `ArchRuby` and validates the detected violations. Occasionally, the software architect can refine the architectural

<sup>8</sup><http://www.dito.com.br>, <http://www.timbeta.com.br>, and <http://metodo.plcadogados.com.br>

rules—which have been specified in step (i)—to avoid false positives. Specifically, we ask the architects to analyze each violation and double check in the source code whether the violation is indeed a true positive. We repeat this process until the architects are confident that the architectural rules indeed represent the system architecture.

- (iii) *Architectural visualization*: The software architect evaluates the reflexion model, one of the high-level architectural models provided by ArchRuby, to better visualize the identified violations. We ask the architects to express an opinion on the readability and representativeness of the reflexion model. Occasionally, to provide a more solid feedback, the architects can share the model with other team members.

### 6.3 Dito Social

**Architectural rules specification:** The software architect specified 62 modules and 43 architectural rules. A relevant subset of the specification is reported in Figure 9.<sup>9</sup> The `dashboard_controller` module is responsible for presenting information to the customers and hence can access several data providers' modules (lines 3–7). The `facebook_info_retriever` module is responsible for retrieving data from Facebook and hence can access only the modules `facebook` and `airbrake` (line 11). The `post_model` is responsible for data persistence and hence must implement classes from module `activerecord` (line 15) and can access modules that provide underlying services (lines 16–18), e.g., `post_workers`. The `report_model` module is responsible for generating reports about posts and interactions, and hence can access modules that provides data and e-mail delivery functionality (e.g., `post_model`, `interaction_model`, `mail`, `aws`, etc.) (line 22).

```

1 dashboard_controller:
2   files: 'app/controllers/dashboard/**/*.rb'
3   allowed: 'dashboard_finder, stats_model, network_model,
4           action_model, app_model, interaction_model, post_model,
5           social_helper, user_network_model, stats_model,
6           controller_base, referral_model, origin_model, http_party,
7           user_agent_model, user_model, airbrake'
8
9 facebook_info_retriever:
10  files: 'lib/facebook_info_retriever.rb'
11  allowed: 'facebook, airbrake'
12
13 post_model:
14  files: 'app/models/post.rb'
15  required: 'activerecord'
16  allowed: 'resque, post_workers, post_logger, facebook_info_retriever,
17          social_helper, interaction_model, question_option_model,
18          logger, activerecord, rails'
19
20 report_model:
21  files: 'app/models/report/**/*.rb'
22  allowed: 'post_model, social_helper, interaction_model, rails, mail, aws, http_party'

```

Figure 9 – Subset of the architectural specification of Dito Social

**Architectural conformance:** ArchRuby could detect 24 violations in Dito Social, as reported in Table 2. Two of these violations are discussed next.

<sup>9</sup>The complete data of the evaluation of the proprietary systems are available at: <http://aserg.labsoft.dcc.ufmg.br/archruby/jot2015>

Table 2 – Architectural violations detected in Dito Social

Module	Rules	# Violations
dashboard_controller	allowed: 'dashboard_finder, ...'	16
dashboard_finder	allowed: 'stats_model, ...'	3
report_model	allowed: 'post_model, ...'	2
event_model	allowed: 'action_model'	1
user_model	allowed: 'user_infos, ...'	1
facebook_info_retriever	allowed: 'facebook, airbrake'	1

*Example of violation #1:* The service of user notification (e.g., e-mail) was moved to another system and hence it is no longer part of Dito Social. Nonetheless, as shown in Figure 10, ArchRuby detected five dependencies (lines 2, 3, 5, and 7) in class `EmailsController`—which belongs to module `dashboard_controller`—to class `Email`, which is not explicitly allowed according to the architectural rules (lines 3–7 of Figure 9). More specifically, class `Email` does not belong to any defined module; in this case, we include such kind of classes in a module called *unknown*.

```

1 def create #from Module dashboard_controller
2   email = Email.new params['email']
3   email.save!
4   send_template_to_mandrill
5   if email.action
6     redis_action_id = SocialHelper::RedisData.get_action_id_by_name
7                       email.action.name, email.app_id
8   end
9 end

```

Figure 10 – Example #1 – Divergence detected in Dito Social

*Example of violation #2:* Module `post_model` is allowed to access module `facebook_info_retriever`, but not the opposite. Nevertheless, as shown in Figure 11, ArchRuby detected two dependencies (lines 15 and 18) in class `FacebookInfoRetriever`—which belongs to module `facebook_info_retriever`—to class `Post` from module `post_model`, which is not allowed according to the architectural rules (line 11 of Figure 9). It is worth noting that our approach could only detect such violation due to our type propagation heuristic, since the type was first inferred in class `Post` (line 5), but it was propagated by the method call to `get_first_likes_comments_and_people` (lines 8-9).

**Architectural visualization:** Figure 12 illustrates a fragment of the reflexion model. We can note divergences (orange edges) from modules `dashboard_controller` (as described in Example #1), `event_model`, `user_model`, `report_model`, and `dashboard_finder` to classes that do not belong to any defined module. We also can note the allowed communication from module `post_model` to `facebook_info_retriever` (black edge). However, the opposite, as described in Example #2, is highlighted as a divergence.

```

1 class Post # from Module post_model
2   def first_update_complete_info_from_facebook(post_info, update_freq,
3     limit = 50, is_customer = false)
4     ...
5     vpost = Post.select('id, fb_id, likes_count, comments_count,
6       updated_info, premium, international').find_by_fb_id(fb_id)
7     facebook = FacebookInfoRetriever.new
8     facebook.get_first_likes_comments_and_people(vpost, limit,
9       special_token.present?) do |info|
10    ...
11  end
12 end
13
14 class FacebookInfoRetriever # from Module facebook_info_retriever
15   def get_first_likes_comments_and_people post, limit = 25,
16     special_token = false, &block
17     ...
18     likes_count = post['likes']
19     ...
20   end
21 end

```

Figure 11 – Example #2 – Divergence detected in Dito Social by type propagation

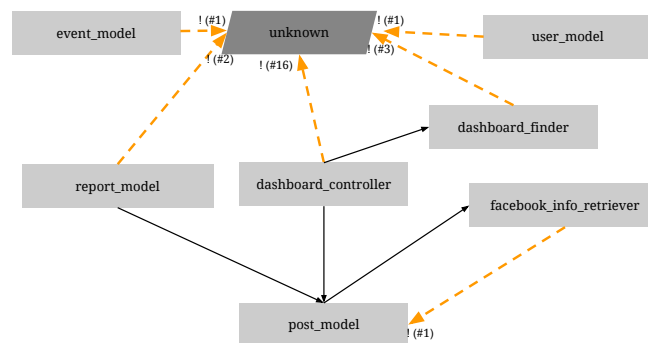


Figure 12 – Fragment of the reflexion model of Dito Social

### 6.3.1 Discussion

The software architect described the architectural rules incrementally. According to the architect, this facilitates the refinement of some rules to avoid false positives in the architecture conformance checking process. It is worth noting that the architect relied on the textual violation report to remember details about the old parts of the system and to refine the architectural rules.

Although the feature responsible for sending e-mail had already been removed from the user interface, it still is in the source code. The architect reported that unused code impacts negatively on the maintainability because it may mislead new developers. Moreover, another critical divergence was found between modules `facebook_info_retriever` and `post_model`. Module `facebook_info_retriever`



is likely to be used with only the Facebook API, i.e., it cannot rely on other parts of the system. According to the architect, this divergence hampers the evolution of the system since module `facebook_info_retriever` is coupled with other parts. Last, the architect argued that, as the number of modules grows, the reflexion model becomes hard to analyze. Particularly in this case study, we also presented the DSM of the system. The architect argue that the two models are complementar, e.g., DSMs are much more appropriate for tasks that require a complete view of the system, but reflexion models are more appropriate to analyze dependencies among few modules.

## 6.4 Tim Beta

**Architectural rules specification:** The software architect specified 43 modules and seven architectural rules. A relevant subset of the specification is reported in Figure 13. Module `models` implements the Model layer of the MVC architectural pattern and hence can access the modules that are responsible for the data persistence (lines 3–7). Module `core` implements the main features of the system and hence must access the modules that provide underlying services (lines 11–14). Module `workers` is responsible for background activities, e.g., updating users’ information based on their facebook profile afterwards they sign in (line 18).

```

1 models:
2   files: 'app/models/**/*.rb'
3   allowed: 'core, helpers, resque, logistica, dito_social_p,
4            postage_app, workers, facebook, devise, csv, olap,
5            twitter_oauth, datapoints, can_can, tim_points, linker,
6            twitter, rails, active_record, image_magick,
7            action_controller'
8
9 core:
10  files: 'app/core/**/*.rb'
11  allowed: 'models, helpers, facebook, twitter, foursquare, gmail,
12           mailers, instagram, dito_social_p, twitter_oauth,
13           contact_us, resque, sanitize, active_record, workers,
14           hoptoad'
15
16 workers:
17  files: 'app/workers/**/*.rb'
18  allowed: 'models, core, facebook, dito_social_p, rails'

```

Figure 13 – Subset of the architectural specification of Tim Beta

**Architectural conformance:** ArchRuby could detect 22 violations in Tim Beta, as reported in Table 3. An example of a detected violation is discussed next.

Table 3 – Architectural violations detected in Tim Beta

Module	Rules	# Violations
core	allowed: 'models, ...'	6
models	allowed: 'core, ...'	16

*Example of violation #3:* Features related to the Orkut social network have been removed from Tim Beta; consequently, the respective source code has been removed as well. Nevertheless, as shown is Figure 14, class `User`—which belongs to module `models`—accesses class `Core::Datapoints::Orkut` (line 2), which is not explicitly

allowed according to the architectural rules (lines 3–7 of Figure 13).

```

1 def update_orkut_stats user_net = nil, app = nil #from Module model
2   orkut_collector = Core::Datapoints::Orkut.new(
3     user_net.access_token,
4     user_net.access_secret,
5     user_net.social_id
6   )
7   orkut_datapoints = orkut_collector.collect
8 end

```

Figure 14 – Example #3 – Divergence detected in Tim Beta

**Architectural visualization:** Figure 15 illustrates a fragment of the reflexion model to better visualize some identified violations. We can note divergences (orange edges) from modules `core` and `models` to classes that do not belong to any defined module; the latter refers to the scenario described in Example #3.

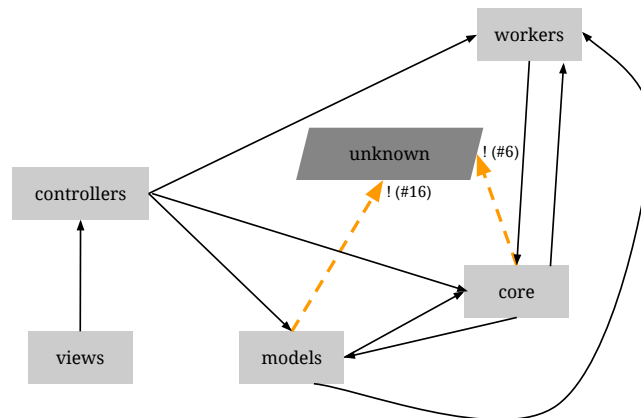


Figure 15 – Fragment of the reflexion model of Tim Beta

#### 6.4.1 Discussion

The architect of `Tim Beta` relied on an artifact that specifies the most important modules in the system as the basis to specify the architectural rules. As a consequence, there is a relative small number of architectural rules. The architectural conformance process detected components that the software architect had thought no longer exists. For instance, all functionality related to the Orkut social network should have been entirely removed from the source code, but they were still found in the source code. Moreover, the architect argue (i) that `ArchRuby` is important to support the architectural monitoring since it is impractical to manually do this process; (ii) that `ArchRuby` should be incorporated into the continuous integration process; and (iii) that the reflexion model can be used by new team developers to understand the system modularization.

## 6.5 PLC Attorneys

**Architectural rules specification:** The software architect specified 14 modules and 11 architectural rules. A relevant subset of the specification is presented in Figure 16. The purpose of the system is to keep the customer aware of the tasks that have been resolved and the ones that are still pendent. Therefore, module `project` is responsible for handling data about the customers' project and can access modules that contain data it needs (line 4). Module `project_relations` is responsible for storing customer data, reporting progress, and displaying charts, and hence can access modules that provide underlying services (line 12). Finally, module `mailers` is responsible for triggering e-mails to clients and can access modules that provide information about the projects (line 17).

```

1 project:
2   files: 'app/models/project.rb'
3   required: 'activerecord'
4   allowed: 'chartdraw, project_relations, admins'
5
6 project_relations:
7   files: 'app/models/area.rb, app/models/company.rb, app/models/areas_project.rb,
8         app/models/attack.rb, app/models/control.rb, app/models/diagnostic.rb,
9         app/models/improvement.rb, app/models/action.rb, app/models/task.rb,
10        app/models/responsible.rb'
11   required: 'activerecord'
12   allowed: 'chartdraw, mailers, admins'
13
14 mailers:
15   files: 'app/mailers/**/*.rb'
16   allowed: 'project_relations, project'
17   required: 'actionmailer'

```

Figure 16 – Subset of the architectural specification of PLC Attorneys

**Architectural conformance:** ArchRuby could detect two violations in PLC Attorneys, as reported in Table 4. We argue that the small number of violations is because the system is small and it is in the beginning of the development, which contributes to developers to commit fewer architectural mistakes. However, we found a critical violation that must be corrected before deploying the system to the production environment. This violation is discussed next.

Table 4 – Architectural violations detected in PLC Attorneys

Module	Rules	# Violations
mailers	required: 'actionmailer'	1
controller	allowed: 'presenters, ...'	1

*Example of violation #4:* The e-mail delivery service relies on Gem `ActionMailer` for the task of sending e-mails. Nevertheless, as shown is Figure 17, class `DiagnosticsMailer`—which belongs to module `mailers`—does not establish dependency with the aforementioned Gem, which is required according to the architectural rules (line 17 of Figure 16). This violation is inevitably critical because the e-mails will not be delivered without the establishment of the dependency with `ActionMailer`.

```

1 class DiagnosticsMailer #from Module mailers
2   default from: "test@test.com"
3
4   def diagnostic_created(admin, project_id, area_id)
5     @admin = admin
6     @project = Project.find(project_id)
7     @area = Area.find(area_id)
8
9     mail(to: @admin.email, subject: '[PLC - Added new diagnostic!']
10  end
11 end

```

Figure 17 – Example #4 – Absence detected in PLC Attorneys

**Architectural visualization:** Figure 18 illustrates a fragment of the reflexion model to better visualize some identified violations. We note an absence (red edge) from module `mailers` to module `actionmailer`, which refers to the scenario described in Example #4. We also note a divergence (orange edge) from module `controller` to `documents`.

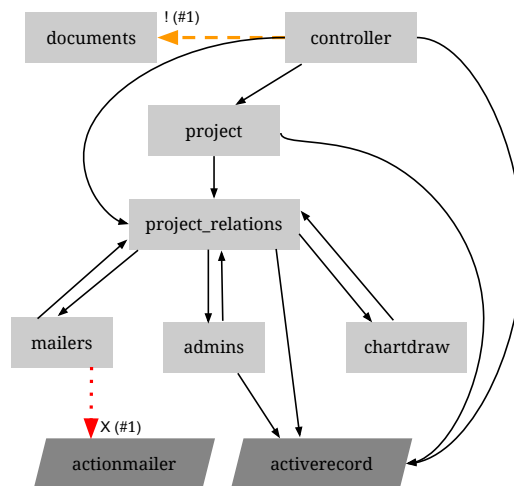


Figure 18 – Fragment of the reflexion model of PLC

### 6.5.1 Discussion

Since the system is in the early stages of its development, the number of architectural rules is relatively small. This also leads to a small number of architectural violations and thereafter to a small number of violations detected by `ArchRuby`. Nevertheless, `ArchRuby` could detect a serious architectural violation. The developer have not established a required dependency between modules `mailers` and `actionmailer`. Without such dependency, the system was unable to send e-mails. The architect reported this violation as a serious one because it breaks a core feature of the system. The architect also suggests that `ArchRuby` should provide means to automatically specify the architectural rules in order to minimize the effort by the software architect.

## 6.6 General Discussion

It is important to highlight some points about the evaluation described in this section: (i) the software architects occasionally had to refine the architectural rules in order to avoid false positives after the architectural conformance process, which indicates that—in practice—the *architectural rules specification* and *architectural conformance* steps are jointly done; (ii) we could detect a high number of divergences in *Dito Social* and *Tim Beta*, which shows that developers establish dependencies with modules that are forbidden (or not explicitly allowed) by the architectural rules; (iii) on the other hand, we could detect few violations in *PLC Attorneys*. Since the system is new and small, we argue that these properties contribute to developers to commit fewer architectural mistakes; (iv) the software architects had no previous knowledge on the identified violations and reported that they negatively impact on the maintenance of the systems; (v) since we rely primarily on reflexion models, the software architects reported issues on visualizing the architectural violations as the number of modules grows, suggesting a scalability problem. In such cases, we allowed the architect to switch the high-level architectural model to DSM; and (vi) the software architects claimed the need for tool support to automatically monitor the source code and perform the architecture conformance checking process.

## 6.7 Threats to Validity

There are two main threats to validity of the study [WRH<sup>+</sup>12]. First, as usual in empirical studies in software engineering, we cannot claim that our approach will provide equivalent results in other systems (external validity). However, we rely on three real-world systems that have been developed by different teams. Second, we relied on three software architects (one per system) to define the rules, to validate the detected violations, and to analyze the visualization model. As typical in human-based classifications, our results might be affected by some degree of subjectivity (construct validity). However, it is important to highlight that we interviewed the software architects who designed the evaluated architectures, and are responsible for their maintenance and evolution. Therefore, they are the right experts to evaluate our proposed approach.

## 7 Effectiveness of the Type Propagation Heuristic

Our type propagation mechanism, as described in Section 4, aims to raise the effectiveness of our approach by increasing the number of analyzed dependencies. In this section—based on the data of our previous evaluation—we provide a quantitative and qualitative discussion on the number and importance of the types inferred by our heuristic (effectiveness).

We observed, in the previous evaluation, that some architectural violations are only detected due to our heuristic. Therefore, we investigated in the three previous evaluated real-world systems—*Dito Social*, *Tim Beta*, and *PLC Attorneys*—the number of types and violations that are only inferred and detected, respectively, due to our heuristic.

### 7.1 Research Questions

We conducted a study to address the following overarching research questions:

**RQ #1** – How many types are only inferred due to our heuristic?

**RQ #2** – How many violations are only detected due to our heuristic?

## 7.2 Data Set

Our study mainly relies on the three previously evaluated real-world systems—namely *Dito Social*, *Tim Beta*, and *PLC Attorneys*—, w.r.t. the number of types and violations that are only inferred and detected, respectively, due to our heuristic.

## 7.3 Results and Discussion

In this section, we provide answers for the proposed research questions.

### 7.3.1 RQ #1: How many types are only inferred due to our heuristic?

Our approach relies on static code analysis techniques to extract the dependencies that should be verified according to the planned architecture. Our heuristic ensures the propagation of the inferred types. Otherwise, only direct inferences of types (e.g., instantiation) would be considered.

**Methodology:** In order to quantify the number of types that are exclusively inferred by our heuristic, we performed *ArchRuby* in the three systems, enabling and disabling our type propagation heuristic. It is important to differentiate (i) the number of *language features*, which refers to expressions, statements, and declarations; (ii) the number of *dependencies*, which refers to every single dependency inspected by the conformance process; and (iii) the number of *inferred types*, which refers to every single triple `[method, var_name, type]` in set `TYPES`, as previously explained in Section 4. Thereupon, the number of language features is far higher than the number of dependencies, which, in turn, is far higher than the number of inferred types. For instance, assume the piece of code in Figure 19. There are seven language features, two dependencies to be inspected by the conformance process (a instantiation of and a method call from class `Test` to type `Z`), and only one inferred type (`[Test::bar, x, Z]`).

```

1  class Test
2    def bar
3      x = Z.new
4      if x.send('foo')
5        y = 3
6      end
7    end
8  end

```

Figure 19 – Number of language features, dependencies, and inferred types

**Results and Discussion:** Table 5 reports our results. On the average, the percentage of additional types is 4.59%. In fact, *PLC Attorneys* was the only system that presented percentagem below 5% since it represents a relative small system in the early stages of its development.

Table 5 – Number of inferred types by our proposed type propagation heuristic

Project	# of inferred types without heuristic	# of inferred types with heuristic	% added
Dito Social	566	598	5.65%
Tim Beta	672	709	5.51%
PLC Attorneys	154	158	2.60%
		<b>Average</b>	<b>4.59%</b>

In order to provide an answer in a broader context, we complementarily replicated our experiment in a data set of 28 open-source Ruby, as described in Appendix B. As reported in Table 6, the percentage of additional types is  $5.03\% \pm 3.95\%$  (average  $\pm$  standard deviation). Statistically, the number of additional types should fall between 3.50% and 6.56% within a 95% confidence interval.

Table 6 – Complementary analysis on the number of inferred types by our heuristic

Project	# of inferred types without heuristic	# of inferred types with heuristic	% added
Active Admin	345	349	1.16%
CanCan	26	26	0.00%
Capistrano	39	39	0.00%
Capybara	155	166	7.10%
CarrierWave	81	85	4.94%
CocoaPods	438	465	6.16%
DevDocs	283	292	3.18%
Devise	114	121	6.14%
diaspora*	934	952	1.93%
Discourse	2,950	3,124	5.90%
FPM	157	172	9.55%
GitLab	1,750	1,794	2.51%
Grape	137	146	6.57%
Homebrew-Cask	426	443	3.99%
Homebrew	8,026	8,125	1.23%
Huginn	463	477	3.02%
Jekyll	259	273	5.41%
Octopress	95	111	16.84%
Paperclip	132	137	3.79%
Rails	2,464	2,559	3.86%
RailsAdmin	231	234	1.30%
Resque	62	68	9.68%
Ruby	4,116	4,391	6.68%
Sass	519	560	7.90%
Simple Form	113	115	1.77%
Spree	1,311	1,324	0.99%
Vagrant	586	620	5.80%
Whenever	15	17	13.33%
		<b>Average</b>	<b>5.03%</b>
		<b>Std Dev</b>	<b>3.95%</b>

After a qualitative analysis of our results, although it is not trivial, we observed that our heuristic could infer more types if it also analyses and propagates the return type of method invocations. For instance, assume the piece of code in Figure 20. In this example, set TYPES would contain the tuples  $[\text{Clazz}::\text{foo}, y, A]$  and

[Clazz::bar, z, B], but if it could infer [Clazz::foo, x, B] by analyzing the returned type, it would also include [A::qux, k, B] (where  $k$  is the name of the formal parameter in A::qux) in the set, which promotes the type propagation through the system.

```

1  class Clazz
2    def foo
3      x = bar()
4      y = A.new
5      y.qux(x)
6    end
7
8    def bar
9      z = B.new
10     z.baz
11     z
12   end
13 end

```

Figure 20 – Potential improvement to the type propagation heuristic

**Final Remarks:** Our heuristic increased the number of inspected dependencies by 5% on the average, but could increase up to 17%. We argue that the increase depends on the underlying programming style. For example, our heuristic achieves better results when developers largely rely on the dependency injection design pattern [Met12].

### 7.3.2 RQ #2: How many violations are only detected due to our heuristic?

The previous research question showed that our heuristic may increase the number of inspected types in 5% on the average. Nonetheless, it is important to investigate whether these additional types contribute to the detection of architectural violations in real scenarios.

**Methodology:** In order to measure the effectivity of our heuristic, i.e., the number of violations that are identified exclusively by our heuristic, we re-performed ArchRuby in the three real-world systems previously evaluated in Section 6.1, enabling and disabling the type propagation heuristic.

**Results and Discussion:** From the 48 architectural violations detected in the three real-world systems, three violations are detected exclusively by our heuristic. Therefore, a preliminary analysis may point out the ineffectiveness of our heuristic. However, we claim that our heuristic was indeed very effective. For example, we found 24 violations on Dito Social. On one hand, from the 566 inferred types without our heuristic, we found 22 violations (3.89%); on the other hand, from the 32 inferred types by our heuristic, we could find two more violations (6.25%). Likewise, we found 22 violations on Tim Beta. On one hand, from the 672 inferred types without the heuristic, we found 21 violations (3.13%); on the other hand, from the 37 inferred types by our heuristic, we could find one more violation (2.70%).

Figure 21 illustrates one of the violations that could be detected exclusively by our heuristic. There are forbidden accesses to class Core::Datapoints::Orkut (lines 7, 9, and 16), which are forbidden since features related to the Orkut social network have been removed from Tim Beta. Specifically for the latest violation (line 16), our approach could only detect it due to our heuristic, since the type



was first inferred in method `verify_orkut_users_friends` (line 7), but it was propagated to the formal parameter `collector` (line 15) through the method call to `check_friends_count` (line 9).

```

1 def self.verify_orkut_users_friends users_ids
2   file = File.open('orkut_log.csv', 'w')
3   orkut_data = Network::ORKUT
4   users_ids.each do |user_id|
5     user_network = UserNetwork.where(:network_id => orkut_data.id,
6                                     :secondary_user_id => user_id).first
7     orkut_collector = Core::Datapoints::Orkut.new(user_network.access_token,
8                                                  user_network.access_secret, user_network.social_id)
9     how_many = check_friends_count(orkut_collector)
10    file.puts "#{user_network.id}, #{user_network.url}, #{how_many}"
11  end
12  file.close
13 end
14
15 def self.check_friends_count(collector)
16   datapoints = collector.collect
17   friends_count = datapoints[:friends]
18   if friends_count > 500
19     #many statements
20   else
21     #few statements
22   end
23 end

```

Figure 21 – Divergence detected in `Tim Beta` using the type propagation heuristic

**Final Remarks:** Some violations are identified exclusively by our heuristic. Disregarding the `PLC Attorneys` system where there are no violations detected by the proposed heuristic, the overall percentage of the violations identified exclusively by our heuristic is 4.35% (3/69), while in the remainder of the system is 3.47% (43/1,238).

## 8 Related Work

We organize related work in three sections: Architecture Conformance Techniques and Tools (Section 8.1); (b) Studies using Dynamic Languages (Section 8.2); and (c) Ruby Tools (Section 8.3).

### 8.1 Architecture Conformance Techniques and Tools

Our solution is inspired by `DCLsuite`, an architectural conformance and repair approach for Java systems [TV09, TVCB15, TV08]. Software architects define a set of constraints using the DCL language, and the tool detects architectural violations and provides suggestions on how to solve them. By contrast, `ArchRuby` targets a dynamic language (which required a definition and implementation of a type propagation heuristic), allows the specification of architectural constraints in YAML files, and provides two high-level architectural models to better visualize the identified violations; however, `ArchRuby` does not contemplate architectural repair.

Several domain-specific languages have been proposed for architecture conformance. SCL (Structural Constraint Language) [HH06] is a Prolog-like DSL to specify and check design intent in C++ or Java code. LogEn [EKKM08] is another Prolog-like language for defining and continuous checking structural dependencies in Java systems. The language has an explicit `type` predicate, where the first argument is a source element and the second parameter is its type name. DesignWizard [BGF11] is an internal DSL for detecting design and architectural anomalies in Java-based software architectures. Developers express the desired architecture by writing tests that make assertions about the structure of the Java code. Therefore, these DSLs are proposed for static languages, typically Java and C++, and heavily depend on type information. Soul [MKPW06] is a Prolog-like language that provides access to the static structure of Smalltalk programs (a dynamic language, like Ruby). However, Soul does not aim to detect architectural violations (divergences and absences), but to enforce source code regularities (e.g., all visitor methods must start with `visit` and must be implemented in a method protocol called `action`).

The high-level architectural model provided by `ArchRuby` is inspired by `SAVE` [KMNL06], an approach based on reflection models [MNS95]. `SAVE` compares (i) the planned architecture, as idealized by the software architect, to (ii) the implemented architecture, as extracted from source code. As a result, the tool computes the reflection model that highlights divergences and absences between these two models. Reflexion models, however, become unreadable as the number of modules and dependencies grows. Thereupon, the other high-level architectural model provided by `ArchRuby` is inspired by `Lattix LDM`, an approach based on DSMs [SJSJ05], which relies on matrices that scale better than graphs. The high-level architectural models produced by `ArchRuby` is very similar to a reflexion model and a DSM, although it is generated from textual architectural rules. More important, developers using `ArchRuby` can decide for the more appropriate high-level model to be used.

`ArchLint` [MVA<sup>+</sup>13, MVT<sup>+</sup>16] proposes a novel approach to detect architectural violations, by mining version repositories. However, the system relies on heuristics to detect patterns of dependencies between modules, which are centered on type information. Inspired on the Z language, `LePUS` [Ede01] is a formal language for specifying object-oriented design and architectural patterns. However, instead of checking architecture intent, `LePUS` focuses on the specification of programming protocols, such as `forward` (that holds when the formal arguments of a method are used to call a method with the same signature) and `produce` (a special kind of `create` in which the new object is used in a return statement). The language also includes a visual version, called `LePUS3` [GNE08].

## 8.2 Studies using Dynamic Languages

Richards et al. [RHBV11] evaluated the use of `eval` in JavaScript, based on a corpus of more than 10,000 popular web sites. Unlike our findings for Ruby, they report that `eval` is popular and do not necessarily harmful. It is usually considered a best practice for specific tasks, such as loading scripts or data asynchronously. The authors also investigated a broad range of JavaScript dynamic features [RLBV10]. They concluded for example that libraries often change the prototype links dynamically, but such changes are restricted to built-in types, such as `Object` and `Array`, and changes in user-created types are more rare.

Hills et al. [HKV13] report a study over a significant corpus of open-source PHP systems to understand how developers actually use dynamic features, including dynamic

file inclusion, handlers for unimplemented methods or fields, an eval expression for executing arbitrary PHP code at runtime, and variadic functions. The authors conclude that a large number of uses of dynamic features could be replaced by static ones without changing the expected behavior. Another group of researchers reached the same conclusion in a case study over a 1,000 systems developed using the Pharo language [CRTR13]. Orru et al. [OTMT15] analyzed a collection of 51 software systems written in Python, to shed light on how they use inheritance. They show that fewer classes inherit from other classes (w.r.t. Java), but more classes are inherited from. They also claim that the dataset used in this study can help to foster further research on Python software [OTM<sup>+</sup>15].

Hanenber et al. [Han10] show that dynamic languages can produce high-quality systems, but requiring less time than when using a static language. They report a laboratory study when groups of developers implemented a lexical analyzer in two versions of a language, with static and dynamic types, respectively. The implementation using a static language required 161 hours, while the implementation using the dynamic version of the language required 108 hours.

### 8.3 Ruby Tools

In the Ruby ecosystem, we are not aware of any architectural conformance and visualization technique as the one proposed in this paper. However, there are several tools that aim to increase the quality of Ruby systems through static code analysis techniques. `Code Climate`<sup>10</sup> is a tool that assists in code reviews. It reports where the system has to improve, e.g., complex methods, security breaches, refactoring opportunities, etc. The tool also suggests reference literature for developers to better understand the problem and the correction actions. `Rubocop`<sup>11</sup> is a Gem that performs static code analysis to verify error and style rules, likewise `LASER`<sup>12</sup> and `ruby-lint`<sup>13</sup>. In another line, `Brakeman`<sup>14</sup> is a security vulnerability scanner and `Pelusa`<sup>15</sup> indicates possible red flags or missing best practices. `ArchRuby`, in turn, complements these tools by providing means to control the architectural erosion process.

## 9 Conclusion

Software architecture erosion is a recurrent problem in software development. Deviations from the planned architecture have strong impact on the system maintainability and evolvability, and may even lead to the rewriting of components. Even more critical, the erosion process might be even more severe in dynamic languages because (i) the dynamic constructs provided by such languages make developers more propitious to break the architecture, and (ii) the developers of dynamic languages lack tool support for monitoring the implemented architecture.

To address these shortcomings, this paper described an architectural conformance and visualization approach based on static code analysis techniques and a lightweight type propagation heuristic. Such a solution provides means to control the architectural erosion process by reporting architectural violations and visualizing them in two

<sup>10</sup><https://codeclimate.com/>

<sup>11</sup><https://github.com/bbatsov/rubocop>

<sup>12</sup><https://github.com/michaeledgar/laser>

<sup>13</sup><https://github.com/YorickPeterse/ruby-lint>

<sup>14</sup><http://brakemanscanner.org/>

<sup>15</sup><https://github.com/codegram/pelusa>

high-level architectural models, namely reflexion models and DSMs. This paper also presented a tool—called ArchRuby—that implements our approach.

We evaluated our solution in three real-world systems identifying 48 architectural violations of which the developers had no prior knowledge. We also evaluated our type propagation heuristic in 28 open-source systems, reporting that (i) the number of analyzed types raises 5% on the average, but it may increase up to 17%; and (ii) certain violations are only detected due to our heuristic. As a practical result, ArchRuby was integrated into the software development process adopted by Dito, the company responsible for the evaluated systems.

Ideas for future work include (i) incorporating an architectural repair solution that provides suggestions on how to solve the detected violations; (ii) improving our type propagation heuristic; (iii) integrating the proposed approach to mainstreams IDEs (e.g., RubyMine) for a better usability; and (iv) extending our solution to other dynamic languages.

The ArchRuby tool and its source code is public available at:

<http://aserg.labsoft.dcc.ufmg.br/archruby>

## References

- [BC99] Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity*. MIT Press, 1999.
- [BGF11] Joao Brunet, Dalton Guerreiro, and Jorge Figueiredo. Structural conformance checking with design tests: An evaluation of usability and scalability. In *27th International Conference on Software Maintenance (ICSM)*, pages 143–152, 2011. doi:10.1109/ICSM.2011.6080781.
- [Bla09] David A. Black. *The Well-Grounded Rubyist*. Manning, 2009.
- [Bor11] Jens Borchers. Invited talk: Reengineering from a practitioner’s view – a personal lesson’s learned assessment. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 1–2, 2011. doi:10.1109/CSMR.2011.63.
- [Bos04] Jan Bosch. Software architecture: The next step. In *First European Workshop (EWSA)*, pages 194–199, 2004. doi:10.1007/978-3-540-24769-2\_14.
- [CRTR13] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. How (and why) developers use the dynamic features of programming languages: the case of smalltalk. *Empirical Software Engineering*, 18(6):1156–1194, 2013. doi:10.1007/s10664-012-9203-2.
- [Ede01] Amnon H. Eden. Formal specification of object-oriented design. In *International Conference on Multidisciplinary Design in Engineering*, pages 256–263, 2001. doi:N/A.
- [EKKM08] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *30th International Conference on Software Engineering (ICSE)*, pages 391–400, 2008. doi:10.1145/1368088.1368142.
- [FhDAFH09] Michael Furr, Jong hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *24th Symposium on Applied*

- Computing (SAC)*, pages 1859–1866, 2009. doi:10.1145/1529282.1529700.
- [GNE08] Epameinondas Gasparis, Jonathan Nicholson, and Amnon H. Eden. LePUS3: An object-oriented design description language. In *5th International Diagrammatic Representation and Inference Conference*, volume 5223 of *Lecture Notes in Computer Science*, pages 19–21. Springer, 2008. doi:10.1007/978-3-540-87730-1\_37.
- [Han10] Stefan Hanenberg. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 22–35, 2010. doi:10.1145/1869459.1869462.
- [HH06] Daqing Hou and H. James Hoover. Using SCL to specify and check design intent in source code. *IEEE Transactions on Software Engineering*, 32(6):404–423, 2006. doi:10.1109/TSE.2006.60.
- [HKV13] Mark Hills, Paul Klint, and Jurgen J. Vinju. An empirical study of PHP feature usage: a static analysis perspective. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 325–335, 2013. doi:10.1145/2483760.2483786.
- [KMHM08] Jens Knodel, Dirk Muthig, Uwe Haury, and Gerald Meier. Architecture compliance checking - experiences from successful technology transfer to industry. In *12th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 43–52, 2008. doi:10.1109/CSMR.2008.4493299.
- [KMNL06] Jens Knodel, Dirk Muthig, Matthias Naab, and Mikael Lindvall. Static evaluation of software architectures. In *10th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 279–294, 2006. doi:10.1109/CSMR.2006.53.
- [KMR08] Jens Knodel, Dirk Muthig, and Dominik Rost. Constructive architecture compliance checking - an experiment on support by live feedback. In *24th International Conference on Software Maintenance (ICSM)*, pages 287–296, 2008. doi:10.1109/ICSM.2008.4658077.
- [Met12] Sandi Metz. *Practical Object-Oriented Design in Ruby: An Agile Primer*. Addison-Wesley, 2012.
- [MKPW06] Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. Co-evolving code and design with intensional views: A case study. *Computer Languages, Systems & Structures*, 32(2-3):140–156, 2006. doi:10.1016/j.cl.2005.09.002.
- [MNS95] Gail Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *3rd Symposium on Foundations of Software Engineering (FSE)*, pages 18–28, 1995. doi:10.1145/222124.222136.
- [MVA<sup>+</sup>13] Cristiano Maffort, Marco Tulio Valente, Nicolas Anquetil, Andre Hora, and Mariza Bigonha. Heuristics for discovering architectural violations. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 222–231, 2013. doi:10.1109/WCRE.2013.6671297.

- [MVT15a] Sergio Miranda, Marco Tulio Valente, and Ricardo Terra. ArchRuby: Conformidade e visualização arquitetural em linguagens dinâmicas. In *VI Brazilian Conference on Software: Theory and Practice (CBSOFT), Tools Session*, pages 1–8, 2015. doi:N/A.
- [MVT15b] Sergio Miranda, Marco Tulio Valente, and Ricardo Terra. Conformidade e visualização arquitetural em linguagens dinâmicas. In *XVIII Ibero-American Conference on Software Engineering (CIbSE), Software Engineering Technologies (SET) Track*, pages 1–14, 2015. doi:N/A.
- [MVT<sup>+</sup>16] Cristiano Maffort, Marco Tulio Valente, Ricardo Terra, Mariza Bigonha, Nicolas Anquetil, and Andre Hora. Mining architectural violations from version history. *Empirical Software Engineering Journal*, pages 1–42, 2016. doi:10.1007/s10664-014-9348-2.
- [OTM<sup>+</sup>15] Matteo Orrú, Ewan Tempero, Michele Marchesi, Roberto Tonelli, and Giuseppe Destefanis. A curated benchmark collection of python systems for empirical studies on software engineering. In *11th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, pages 1–4, 2015. doi:10.1145/2810146.2810148.
- [OTMT15] Matteo Orru, Ewan Tempero, Michele Marchesi, and Roberto Tonelli. How do Python programs use inheritance? a replication study. In *22nd Asia Pacific Software Engineering Conference (APSEC)*, pages 19–21, 2015. doi:N/A.
- [Par94] David Lorge Parnas. Software aging. In *16th International Conference on Software Engineering (ICSE)*, pages 279–287, 1994. doi:10.1109/ICSE.1994.296790.
- [PTD<sup>+</sup>10] Leonardo Passos, Ricardo Terra, Renato Diniz, Marco Tulio Valente, and Nabor Mendonça. Static architecture-conformance checking: An illustrative overview. *IEEE Software*, 27(5):82–89, 2010. doi:10.1109/MS.2009.117.
- [RHBV11] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *25th European Conference on Object-oriented Programming (ECOOP)*, pages 1–27, 2011. doi:10.1007/978-3-642-22655-7\_4.
- [RLBV10] Gregor Richards, Sylvain Lebesne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *31st Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 2010. doi:10.1145/1806596.1806598.
- [SGCH01] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *9th International Symposium on Foundations of Software Engineering (FSE)*, pages 99–108, 2001. doi:10.1145/503209.503224.
- [SJSJ05] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *20th Conference on Object-Oriented Programming, Systems, Languages,*

- and Applications (OOPSLA)*, pages 167–176, 2005. doi:10.1145/1094811.1094824.
- [SRK<sup>+</sup>09] Santonu Sarkar, Shubha Ramachandran, G. Sathish Kumar, Madhu K. Iyengar, K. Rangarajan, and Saravanan Sivagnanam. Modularization of a large-scale business application: A case study. *IEEE Software*, 26:28–35, 2009. doi:10.1109/MS.2009.42.
- [TV08] Ricardo Terra and Marco Tulio Valente. Towards a dependency constraint language to manage software architectures. In *2nd European Conference on Software Architecture (ECSA)*, pages 256–263, 2008. doi:10.1007/978-3-540-88030-1\_19.
- [TV09] Ricardo Terra and Marco Tulio Valente. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 32(12):1073–1094, 2009. doi:10.1002/spe.931.
- [TVCB15] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki, and Roberto S. Bigonha. A recommendation system for repairing violations detected by static architecture conformance checking. *Software: Practice and Experience*, 45(3):315–342, 2015. doi:10.1002/spe.2228.
- [WRH<sup>+</sup>12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer, 2012. doi:10.1007/978-3-642-29044-2.







## B Open-source Data Set

Table 7 summarizes information about our open-source data set. It contains 28 out of the 30 most starred Ruby projects in Github (on September, 2015), which represents a large and heterogeneous collection of software systems, ranging from management systems and remote server automation to frameworks and medium-sized general-purpose libraries.<sup>16</sup> We discarded only two projects: **Bootstrap for Sass**, a tiny project that solely provides support to the Sass-based bootstrap CSS framework; and **Software Engineering Blogs**, which is not an application but a plain Ruby script that generates an OPML<sup>17</sup> file with a list of technology web sites. In total, we analyzed over half million LOC and eight thousand `rb` files.

Table 7 – Evaluated open-source systems

Project and version	LOC	# of rb files	# of Gems
Active Admin (v1.0.0.pre1)	6,053	154	42
CanCan (v1.6.10)	878	16	13
Capistrano (v3.4.0)	2,544	44	7
Capybara (v2.5.0)	8,894	107	20
CarrierWave (v0.10.0)	2,075	37	15
CocoaPods (v0.39.0.beta.4)	8,128	94	41
DevDocs (66cefbd)	12,339	293	27
Devise (v3.4.1)	3,007	60	19
diaspora* (v0.5.2.0)	6,775	126	128
Discourse (vlatests-realease)	14,183	219	101
FPM (v1.4.0)	3,537	25	11
GitLab (v7.14.1)	11,591	219	137
Grape (v0.13.0)	3,370	88	24
Homebrew-Cask (v0.56.0)	5,720	136	8
Homebrew (8278b89)	133,322	3,429	4
Huginn (f4b8e73)	1,464	18	90
Jekyll (v3.0.0.pre.beta8)	3,911	61	39
Octopress (v2.0)	1,313	23	13
Paperclip (v4.3.0)	3,081	59	34
Rails (v4.2.4)	55,530	849	82
RailsAdmin (v0.7.0)	4,624	111	48
Resque (v1.25.0.pre)	1,885	25	12
Ruby (v2_2_3)	170,345	1,076	0
Sass (v3.4.18)	13,080	130	8
Simple Form (v3.1.0.rc2)	2,007	55	9
Spree (v3.0.4)	5,947	149	6
Vagrant (v1.7.4)	8,156	126	21
Whenever (v0.9.4)	632	13	3

<sup>16</sup><https://github.com/search?l=ruby&p=1&q=stars%3A%3E1&s=stars&type=Repositories>, as available on August 2015.

<sup>17</sup>Outline Processor Markup Language (OPLM) is an XML format for outlines, which is straightforward imported by RSS readers.

## About the authors



**Sergio Miranda** is a M.Sc. student in Computer Science at Federal University of Minas Gerais, Brazil. He has been team leader in Dito IT company for six years, working with scalability and architecture of large-scale systems. Contact him at [sergio.miranda@dcc.ufmg.br](mailto:sergio.miranda@dcc.ufmg.br).



**Elder Rodrigues Jr** is an undergraduate student of Computer Science at Federal University of Lavras, Brazil. He also received the technical high school degree in Informatics from CEFET-MG, Brazil (2012). He has worked as teaching assistant of Algorithms and Data Structures for one year. Currently, he holds a CNPq junior research grant to work on architectural conformance. Contact him at [elderjr@computacao.ufla.br](mailto:elderjr@computacao.ufla.br).



**Marco Tulio Valente** received his PhD degree in Computer Science from the Federal University of Minas Gerais, Brazil (2002), where he is an assistant professor in the Computer Science Department, since 2010. His research interests include software architecture and modularity, software maintenance and evolution, and software quality analysis. He is a “Researcher I-D” of the Brazilian National Research Council (CNPq). He also holds a “Researcher from Minas Gerais State” scholarship, from FAPEMIG. Valente has co-authored more than 80 refereed papers in international conferences and journals. Currently, he heads the Applied Software Engineering Research Group (ASERG), at DCC/UFMG. Contact him at [mtov@dcc.ufmg.br](mailto:mtov@dcc.ufmg.br), or visit [www.dcc.ufmg.br/~mtov](http://www.dcc.ufmg.br/~mtov).



**Ricardo Terra** received his Ph.D. degree in Computer Science from Federal University of Minas Gerais, Brazil (2013) with a 1-year internship at the University of Waterloo, Canada. Since 2014, he is an assistant professor in the Department of Computer Science at Federal University of Lavras, Brazil. His research interests include software architecture maintainability and evolvability. Contact him at [terra@dcc.ufla.br](mailto:terra@dcc.ufla.br), or visit [www.dcc.ufla.br/~terra](http://www.dcc.ufla.br/~terra).

**Acknowledgments** Our research has been supported by CAPES, FAPEMIG, and CNPq.