ARCHRUBY: CONFORMIDADE ARQUITETURAL EM LINGUAGENS DINAMICAMENTE TIPADAS

SÉRGIO HENRIQUE MIRANDA JÚNIOR

ARCHRUBY: CONFORMIDADE ARQUITETURAL EM LINGUAGENS DINAMICAMENTE TIPADAS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Marco Túlio de Oliveira Valente
Coorientador: Ricardo Terra Nunes Bueno Villela

Belo Horizonte

Março de 2017

SÉRGIO HENRIQUE MIRANDA JÚNIOR

ARCHRUBY: ARCHITECTURE CONFORMANCE CHECKING IN DYNAMICALLY TYPED LANGUAGES

Dissertation presented to the Graduate Program in Computer Science of the Universidade Federal de Minas Gerais – Departamento de Ciência da Computação in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Marco Túlio de Oliveira Valente
Co-Advisor: Ricardo Terra Nunes Bueno Villela

Belo Horizonte
March 2017

Miranda Júnior, Sérgio Henrique

M672a ArchRuby: Architecture Conformance Checking in Dynamically Typed Languages / Sérgio Henrique Miranda Júnior. — Belo Horizonte, 2017
xxi, 89 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais – Departamento de Ciência da Computação
Orientador: Marco Túlio de Oliveira Valente
Coorientador: Ricardo Terra Nunes Bueno Villela

1. Computação – Teses. 2. Engenharia de software. 3. Linguagem dinamicamente tipada. 4. Erosão arquitetural. 5. Architecture conformance checking. I. Orientador. II. Coorientador. III. Título.

CDU 519.6*32(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Archruby: architecture conformance checking in dynamically typed languages

SÉRGIO HENRIQUE MIRANDA JÚNIOR

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

Prof. Marco Túlio de Oliveira Valente - Orientador
Departamento de Ciência da Computação - UFMG

[illegible]

Prof. Ricardo Terra Nunes Bueno Villela - Coorientador
Departamento de Ciência da Computação - UFLA

Eduardo Figueiredo

Prof. Eduardo Magno Lages Figueiredo
Departamento de Ciência da Computação - UFMG

[illegible]

Prof. Rafael Serapilha Durelli
Departamento de Ciência da Computação - UFLA

Belo Horizonte, 29 de março de 2017.

Agradecimentos

Agradeço imensamente a minha família. Especialmente, agradeço meus pais pelo exemplo de garra e determinação para se alcançar o que deseja, vocês foram um espelho para mim durante toda a caminhada. Agradeço também ao meu irmão, pelo ser humano incrível que é e pela força de vontade que tem. Vocês são o meu alicerce.

Agradeço a minha namorada, amiga, e companheira de vida Débora, que sempre está ao meu lado, me incentivando e me apoiando em todos os momentos. Com você ao meu lado a vida é mais bela.

Agradeço ao meu orientador Marco Túlio, por todo ensinamento. Foi um grande aprendizado trabalhar ao lado desse excelente professor. Seu conhecimento, energia e profissionalismo foram essenciais para este trabalho.

Agradeço ao meu coorientador Ricardo Terra, que foi muito mais do que coorientador neste trabalho. Sua energia, conhecimento e vontade de fazer o melhor tiveram uma imensa contribuição para este trabalho. Mais do que isso, deixo registrado aqui a minha gratidão por toda nossa caminhada juntos, aprendi e aprendo muito com você.

Agradeço aos amigos e familiares por entenderem minha ausência em vários momentos. Mais do que isso, agradeço o incentivo dado por todos também.

Agradeço aos amigos do grupo de pesquisa ASERG pela convivência e troca de conhecimento.

Por fim, agradeço ao Programa de Pós-Graduação em Ciência da Computação (PPGCC) pelo excelente curso e toda atenção prestada durante a construção deste trabalho.

Resumo

Erosão arquitetural é um problema recorrente na evolução de software. Esse problema se agrava em sistemas desenvolvidos em linguagens dinamicamente tipadas devido (i) a certos recursos providos por tais linguagens tornarem os desenvolvedores mais propícios a quebrar a arquitetura planejada, e (ii) a comunidade de desenvolvedores sofrer da falta de ferramentas para análise de arquiteturas. Assim, esta dissertação de mestrado propõe uma solução de conformidade e visualização arquitetural baseada em técnicas de análise estática de código e em uma heurística de inferência de tipos para linguagens dinamicamente tipadas. A ideia central é prover à comunidade de desenvolvedores formas de controlar o processo de erosão arquitetural através da detecção de violações arquiteturais e da visualização de um modelo de alto nível da arquitetura implementada, na forma de modelos de reflexão e DSMs. Nesse sentido, foi projetada uma ferramenta, chamada ArchRuby, que implementa a solução proposta. Para avaliar tal solução, foram realizadas quatro avaliações. Primeiro, a solução proposta foi avaliada em três sistemas reais, sendo capaz de identificar 48 violações arquiteturais das quais os arquitetos não tinham conhecimento. Segundo, foi avaliada a acurácia da heurística de inferência de tipos, concluindo-se que (i) a quantidade de tipos analisados aumenta em 5% na média e (ii) certas violações só foram identificadas devido a essa heurística. Terceiro, realizou-se um estudo para comparar a heurística de inferência de tipos proposta com técnicas de análise dinâmica de código, concluindo-se que (i) a heurística de inferência de tipos proposta provê uma revocação média de 44% e (ii) sete melhorias podem ser incorporadas em técnicas de análise estática de código para aumentar a quantidade de tipos inferidos. Quinto e último, realizou-se um estudo com um sistema real adaptando a ferramenta ArchRuby para utilizar informações geradas por técnicas de análise dinâmica de código, a fim de aumentar o número de dependências a serem analisadas.

Palavras-chave: Conformidade arquitetural; Erosão arquitetural; modelos arquiteturais de alto nível; linguagens dinamicamente tipadas.

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 prone to break the planned architecture (e.g., dynamic invocations and buildings), and (ii) the developers community lacks tool support for monitoring the implemented architecture. To address these shortcomings, we propose an architectural conformance and visualization approach based on static code analysis techniques and on a type inference heuristic to address the particularities of dynamically typed languages. 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. We also describe a tool—called ArchRuby—that implements our approach. To evaluate the proposed approach we conducted four evaluations. First, we evaluate our solution in three real-world systems identifying 48 architectural violations of which the developers had no prior knowledge. Second, we measure the effectiveness of our type inference heuristic reporting that (i) the number of analyzed types raises 5% on average and (ii) certain violations are only detected due to this heuristic. Third, we conducted a study to compare the proposed type inference algorithm with dynamic techniques showing that (i) the proposed type inference algorithm provides an average recall of 44% and (ii) seven improvements can be implemented by static techniques to raise the number of inferred types. Fourth and last, we conduct a study with a real-world system adapting ArchRuby to use information generated by dynamic analysis in order to raise the number of analyzed dependencies.

Palavras-chave: Architecture conformance checking; architectural erosion; high-level architectural models; dynamically typed languages.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

In this chapter, we outline the problem we investigate in this master dissertation. Section 1.1 presents the motivation of this work. Next, Section 1.2 presents an overview of our approach. Section 1.3 shows the structure of this master dissertation, including the organization of its chapters. Finally, Section 1.4 presents our publications.

1.1 Motivation

Software systems should be planned before they get implemented. For this reason, architectural design is a crucial software development activity. Software architects, during the architectural design phase, are concerned with defining how a system is organized and also designing the overall structure of the system [Sommerville, 2010]. For this reason, architectural design is one of the first tasks performed in the software development process [Sommerville, 2010]. Moreover, this design can directly affect software performance, robustness, portability, maintainability, since it comprises a set of key decisions and best practices that enable software evolution [Passos et al., 2010; Murphy et al., 1995].

In this context, after the implementation of a software system, there are two architectures: the concrete architecture (i.e., the architecture followed by source code) and the planned one. Additionally, the concrete architecture usually deviates from the planned one, nullifying the benefits provided by an architectural design [Passos et al., 2010]. In other words, the concrete and the planned architectures are not always aligned. This phenomenon is known as software architecture erosion and it is considered a challenging research problem in software architecture [Knodel et al., 2008a; Terra et al., 2015; Sarkar et al., 2009; Borchers, 2011; Bosch, 2004].

To tackle this problem, several techniques have been proposed [Murphy et al., 1995; Sullivan et al., 2001; Terra and Valente, 2009; Maffort et al., 2013]. Basically, in all these techniques, the software architect needs to define the planned architecture that is used to compare with the concrete one. Moreover, most existing tools that support such techniques rely on static analysis to map the source code to a high-level architectural model [Knodel et al., 2006; Terra and Valente, 2009; Sullivan et al., 2001]. Usually, such techniques and tools depend on information that can be easily extracted from the source code of programs implemented in statically typed languages since these languages enforce developers to declare the types they use in the program. This explicit declaration of types facilitate the identification of the dependencies that are established in the software, which is a key information to any architectural conformance process.

Currently, dynamically typed languages are steadily growing in importance and usage [TIOBE index, 2017]. In that sense, developers that use these languages should also benefit from architectural conformance techniques. Nevertheless, none of the existing techniques and tools to this purpose addresses the particularities of dynamically typed languages. For example, dynamically typed languages only type check the program at run time, which is different from static languages that enforce type constraints at compile type, i.e., without executing the program [Agesen et al., 1995; Agesen and Holzle, 1995a]. Therefore, to extract type information with static analysis techniques from dynamically typed languages, it is necessary type inference algorithms.

1.2 An Overview of the Proposed Approach

As described in the previous section, the concrete and the planned architectures of a software system are not always aligned. Several techniques have been proposed to reveal software architecture violations, but none of them addresses the particularities of dynamically typed languages. Moreover, developers that use dynamically typed languages also need to control the architectural erosion problem. For example, Table 1.1 reports size information about the Top-5 Ruby systems with more stars on GitHub.[1] We claim that due to their large size (i.e., three out of five systems have more than 100 KLOC) and the high number of contributors (i.e., legacy-homebrew has more than five thousand contributors), these systems are also likely to suffer from the architectural erosion problem. To tackle this problem, we describe in this master dissertation an approach to perform architectural conformance and to better visualize the architecture

[1] `http://github.com/search?l=ruby&p=1&q=stars%3A%3E1&s=stars&type=Repositories`, as available on January 2017.

Table 1.1: Top 5 Ruby systems with more stars on GitHub

System	**LOC**	**# of contributors**
Rails (v5.0.1)	212,239	3,223
legacy-homebrew (5a9e19f)	16,656	5,636
Jekyll (v3.4.0)	13,127	675
Discourse (v1.8.0.beta4)	124,487	559
Gitlabhq (v8.16.3)	198,771	1,189

of systems implemented in dynamically typed languages. Specifically, our approach targets systems implemented in Ruby.

As illustrated in Figure 1.1, the proposed approach receives as input the architectural rules (in1) and the source code (in2) of a target system implemented in Ruby. After parsing the architectural rules file (t1) and the source code (t2), the proposed approach triggers the architectural conformance process (t3) in order to detect design decisions that do not respect the intended architecture. To this purpose, the proposed approach uses a type inference algorithm to raise the number of dependencies to be analyzed in the conformance process.

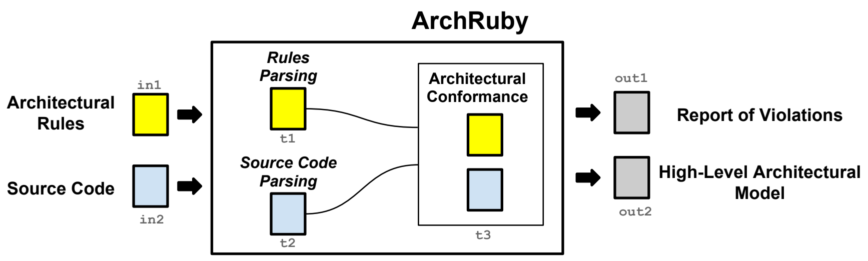


Figure 1.1: The proposed approach

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 distinguish the dependencies that represent violations. It is important to mention that we have also designed a Domain Specific Language (DSL) to specify architectural rules. For example, assume we have a system where a module called `fetcher` is allowed to communicate with a module `twitter_api`. The following code illustrates the specification of architectural constraints for `fetcher`:

```
fetcher:
  files:  'lib/fetcher.rb'
  allowed: 'twitter_api'
```

In line 1, we define the module name; in line 2, we define the files that compose the module; and, in line 3, we define that `fetcher` is allowed to depend only on module `twitter_api`.

To evaluate our approach, we designed a prototype tool, called `ArchRuby`, which is a command-line tool that implements the approach proposed in this dissertation. Our evaluation relies on three real-world systems—namely `Dito Social`, `Tim Beta`, and `PLC Attorneys`—and, `ArchRuby` could find 48 architectural violations developers had no prior knowledge. Moreover, the proposed solution also provides developers with two high-level views of the software architecture that help them to reason about the system organization.

We also measure the effectiveness of the proposed type inference heuristic reporting that (i) the number of analyzed types raises 5% on average and (ii) certain violations are only detected due to this heuristic. Moreover, we conducted a study to compare the proposed type inference algorithm against dynamic techniques showing that (i) the proposed algorithm provides an average recall of 44% and (ii) seven improvements could be implemented in the proposed algorithm to raise the number of inferred types. Finally, we conducted another study with a real-world system reporting that information generated by dynamic techniques could be used by `ArchRuby` to raise the number of analyzed dependencies.

1.3 Outline of the Dissertation

We organized this master dissertation as follows:

- **Chapter 2** discusses background work related to this dissertation. It covers an introduction about architectural conformance and a discussion about architectural conformance techniques. We also discuss about dynamic languages, introducing some historical arguments and key features. In this context, we also provide an overview of Ruby features. Finally, we discourse about the type inference problem.

- **Chapter 3** presents the proposed architectural conformance approach, including a running example that illustrates its operation. This chapter also presents the rules used to define the system architecture and details about architectural conformance process. It shows two high-level models proposed to better visualize the system architecture and describes the proposed type inference heuristic. Finally, we described a tool, called `ArchRuby`, that implements the proposed approach.

- **Chapter 4** evaluates our approach in three systems—named `Dito Social`, `Tim Beta`, and `PLC Attorneys`. In this evaluation, we first asked the architects to define the architectural rule specification of each system. Second, the architects performed our architectural conformance process. Third and last, they analyzed the reported architectural violations and the high-level architectural visualizations generated by `ArchRuby`. More important, quantitative and qualitative discussions are conducted for each system.

- **Chapter 5** presents a study conducted to evaluate the proposed type inference heuristic. We analyzed the following aspects regarding the proposed heuristic: number of inferred types, number of detected violations, accuracy when compared with dynamic techniques, and improvements that can be implemented to raise the number of inferred types. Furthermore, we also study the impact of adding type information as generated by dynamic techniques to `ArchRuby`.

- **Chapter 6** presents the final considerations of this master dissertation, including the contributions, limitations, and future work.

1.4 Publications

This dissertation generated the following publications and therefore contais material from them:

- Miranda, S., Valente, M. T., and Terra, R. (2015b). Conformidade e visualização arquitetural em linguagens dinâmicas. In *XVIII Ibero-American Conference on Software Engineering (CIbSE), Software Engineering Technologies (SET) Track*, pages 137–150. (***Best paper award***)
- Miranda, S., Valente, M. T., and Terra, R. (2015a). ArchRuby: Conformidade e visualização arquitetural em linguagens dinâmicas. In *VI Brazilian Conference on Software: Theory and Practice (CBSoft), Tools Session*, pages 17–24. (***Third best tool award***)
- Miranda, S., Rodrigues, E., Valente, M. T., and Terra, R. (2016a). Architecture conformance checking in dynamically typed languages. *Journal of Object Technology*, 15(3):1–34.
- Miranda, S., Valente, M. T., and Terra, R. (2016b). Inferência de tipos em ruby: Uma comparação entre técnicas de análise estática e dinâmica. In *IV Workshop on Software Visualization, Evolution and Maintenance (VEM)*, pages 105–112. (***Best paper award***)

Chapter 2

Background

In this chapter, we discuss background work related to this master dissertation. Section 2.1 describes what is software architecture conformance. Section 2.2 introduces basic concepts about architecture conformance techniques, which are essential to the understanding of our work. Section 2.3 presents an overview on dynamically typed languages. Section 2.4 describes some Ruby features since our approach is implemented in Ruby. Section 2.5 discusses type inference because it can help to improve the architectural conformance process when performed in dynamically typed languages. Finally, Section 2.6 concludes with a general discussion.

2.1 Architectural Conformance

Architecture design is a crucial software development activity. It is concerned with defining how a system should be organized and designing the overall structure of that system [Sommerville, 2010]. For this reason, it is one of the first tasks performed in the software design process [Sommerville, 2010]. In this context, it is also a critical link between software description and software design. Moreover, the architecture design can directly affect software performance, robustness, portability, maintainability, since it comprises a set of standards and best practices that enable the software evolution [Passos et al., 2010; Murphy et al., 1995]. The expected output of this process is an architectural model that describes how the system should be organized.

However, as the project evolves—due to lack of knowledge, short deadlines, etc.—the defined architectural patterns tend to deteriorate and therefore nullify the benefits provided by an architectural design [Passos et al., 2010]. This phenomenon is known as software architecture erosion and it is considered a challenging research problem in software architecture [Knodel et al., 2008a; Terra et al., 2015; Sarkar et al., 2009;

Borchers, 2011; Bosch, 2004]. To tackle this problem, many architectural conformance techniques have been proposed.

Basically, architectural conformance is the process that checks to which degree the concrete architecture (e.g., the source code implementation) is consistent with the planned architecture of a system [Knodel et al., 2008b]. Architectural conformance can be performed statically (i.e., without executing the target system) or dynamically (i.e., executing the target system). On one hand, static architectural conformance techniques are non-invasive and depend only on the source code. Hence, they do not impact the developer programming activities or the system execution. On the other hand, techniques based on dynamic analysis are performed during the system execution. Therefore, they work best with systems whose behavior may change at run time, such as the ones that make heavy use of dependency injection, reflexion, and meta-programming.

To illustrate the architectural conformance process and its terms we use a hypothetical system that contains the following classes: `Professor`, `Student`, and `Course`. Moreover, suppose that these classes are implemented in files `professor.rb`, `student.rb`, and `course.rb`, respectively. Also, suppose that the architect defines the following modules: `professor` (composed by `professor.rb` file), `student` (composed by `student.rb` file), and `course` (composed by `course.rb` file). Listing 2.1 presents the code fragment of the classes mentioned earlier.

```
class Professor       class Student           class Course
  ...                   ...                     ...
  def add_to_dis(cor)   def add_to_dis(cor)     def add_prof(prof)
    cor.add_prof(self)    cor.add_student(self)   @prof.push(prof)
  end                   end                     end
  ...                   ...
end                   end                       def add_student(stud)
                                                  @students.push(stud)
                                                end
                                                ...
                                               end
```

Listing 2.1: Piece of code to illustrate the architectural conformance process

In order to check architectural conformance, the architect has to define the planned architecture of the system. Suppose that the planned architecture is defined as follows: module `professor` must depend on module `employee`, module `course` must depend on modules `professor` and `student`, modules `professor` and `student` must not depend on module `course`. After that, it is possible to check if the source code implementation respects the rules defined by the planned architecture.

In this scenario two divergencies—dependencies that exist in the source code but that are not prescribed by the planned architecture [Passos et al., 2010]—are detected by the architectural conformance process. The planned architecture prescribed that modules `professor` and `student` must not depend on module `course`, but, the `Professor` and `Student` classes receive an instance of `Course` class (refer to Listing 2.1, line 4). Moreover, an absence—dependencies that do not exist in the source code but that are required by the planned architecture [Passos et al., 2010]—are reported in the `professor` module. The planned architecture prescribed that this module should depend on module `employee`, but such dependency does not exist in the source code (refer to Listing 2.1, class `Professor`).

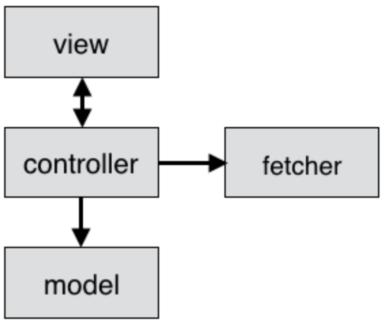
2.2 Architectural Conformance Techniques

To avoid the software architecture erosion phenomenon, several techniques have been proposed. Since our approach works on architecture violations detected by a static architecture conformance process, an overview on architecture conformance techniques is relevant. Specifically, this section provides an overview of the following state-of-the-art techniques:

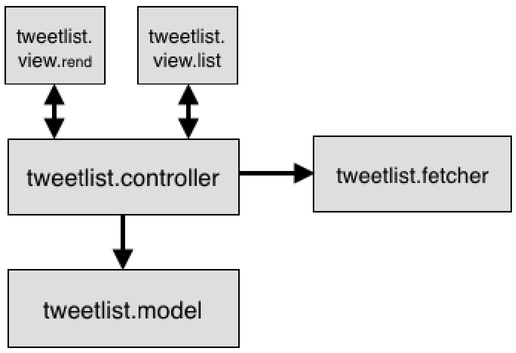
Reflexion Models (RMs): As proposed by Murphy et al. [Murphy et al., 1995], Reflexion Models compare two models. One representing a high-level model of the system (e.g., specified by the developer) and another representing the low-level model (e.g., produced either by statically analyzing the system source or by collecting information during the system execution). Specifically, the architect must define a high-level model that characterizes the planned architecture. Next, a tool is used to extract structural information from the concrete implementation. This extraction can be a call graph, event interactions, etc. Finally, the architect defines a mapping between these two models. Once concluded this last task, the technique computes a software reflexion model that matches the high-level and the low-level models. Additionally, the software architect can modify the high-level or the low-level models to iteratively refine the reflexion models.

The reflexion model highlights divergences and absences, regarding the compared models. Divergences indicate source code interactions that are not expected by the planned architecture and absences indicate interactions that are expected but that are not found. The outcome of the evaluation is summarized and documented in a separated report, which is presented as a graph or text. The former connects the system modules and reports the detected divergences and absences. More specifically, the solid lines in the graph, called convergences, indicate source code interactions that are expected by the planned architecture. On the other hand, divergences are illustrated with dashed lines and absences with dotted lines.

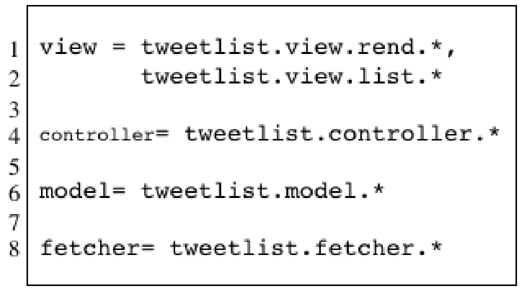
To illustrate the reflexion model technique, we assume a hypothetical system called `TweetList` following a MVC architecture with modules *model*, *view*, *controller* (as expected in this type of architecture), and another module *fetcher*, which is responsible to communicate with the Twitter API. Hence, in this architecture, module *view* is allowed to establish dependency only with module *controller*. Figure 2.1 illustrates the steps that the architect must follow and the output computed by the Reflexion Model technique for the `TweetList` system.



(a) High-level model



(b) Source code model



```
view = tweetlist.view.rend.*,
       tweetlist.view.list.*

controller= tweetlist.controller.*

model= tweetlist.model.*

fetcher= tweetlist.fetcher.*
```

(c) Mapping

Figure 2.1: Reflexion model example

The architect needs to specify the high-level model, to extract the source code model, and to provide a mapping between them (Figures 2.1a, 2.1b, and 2.1c, respectively). The technique then computes the reflexion model that matches the high and low-level models, as illustrated in Figure 2.2.

There are many tools that automate the Reflexion Model technique. In this example, we use the SAVE tool [Knodel et al., 2006]. Assume again the aforementioned MVC architecture. SAVE provides automatic extraction of the implemented source code, i.e., the concrete architecture of the target system. Next, the architect must define a declarative mapping between the concrete architecture and the high-level model. Finally, SAVE computes the reflexion model by comparing the high-level model and the concrete architecture.

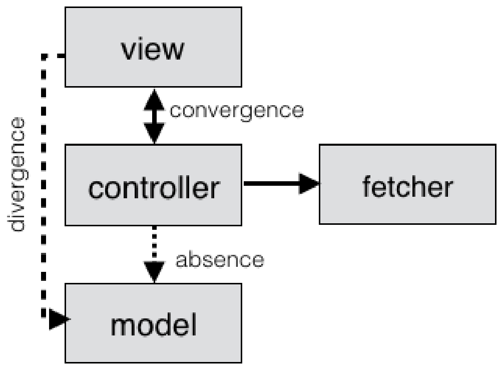
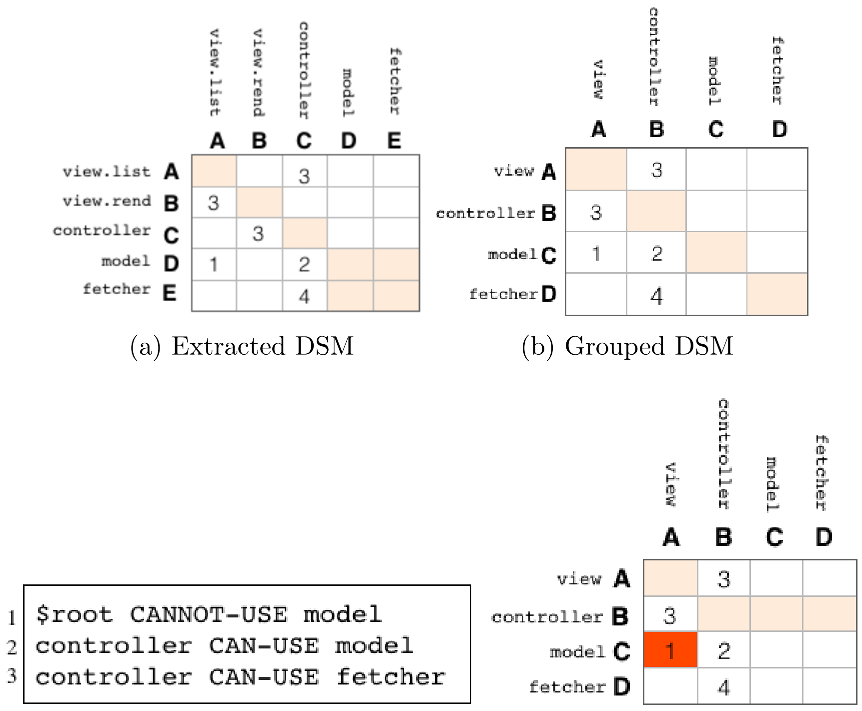


Figure 2.2: Reflexion model

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 [Baldwin and Clark, 1999]. Thereafter, Sullivan et al. claimed that DSM could also be used in software industry [Sullivan et al., 2001]. A DSM is a square matrix where the rows and columns represent the modules of the system. Traditionally, DSM uses 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 [Sangal et al., 2005]. Hence, it is simple to reason about dependencies established between modules since the number of references is explicit written in the cells. In LDM tool, it is possible to distinguish the violations 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. Therefore, design rules can be used to specify architectural patterns such as layering, componentization, external library usage, and other dependency patterns between subsystems. Finally, DSM has a more scalable output than the output generated by reflexion models, since matrices scale better than graphs.

In this illustrative example, we rely on DSM as computed by LDM tool. We assume again the MVC architecture defined in Figure 2.1a. Figure 2.3a illustrates the DSM that is automatically extracted from the source code. Basically, package `tweetlist.view.list` depends on packages `tweetlist.view.rend` and `tweetlist.controller`, package `tweetlist.view.rend` depends on package `tweetlist.controller`, and package `tweetlist.controller` depends on `tweetlist.view.list`, `tweetlist.model`, and `tweetlist.fetcher`. It is also possible to group and rename the packages to improve the visualization. Figure 2.3b illustrates the DSM after grouping packages `tweetlist.view.rend` and `tweetlist.view.list` into a module named *view*. Additionally, packages `tweetlist.controller` and `tweetlist.model` are grouped into modules *controller* and *model*, respectively.



		view.list A	view.rend B	controller C	model D	fetcher E
view.list	A			3		
view.rend	B	3				
controller	C		3			
model	D	1		2		
fetcher	E			4		

(a) Extracted DSM

		view A	controller B	model C	fetcher D
view	A		3		
controller	B	3			
model	C	1	2		
fetcher	D		4		

(b) Grouped DSM

```
1 $root CANNOT-USE model
2 controller CAN-USE model
3 controller CAN-USE fetcher
```

(c) Design rules

		view A	controller B	model C	fetcher D
view	A		3		
controller	B	3			
model	C	🟥 1	2		
fetcher	D		4		

(d) Checked DSM

Figure 2.3: DSM example

In order to check the architectural conformance, architects should define design rules as mentioned before. For instance, Figure 2.3c shows a design rule that allows only *controller* to depend on *model* (lines 1-2). First, this rule specifies that `$root`, which denotes all types of the system, cannot access services provided by *model* (line 1). Next, an exception to the previous rule is defined, specifying that classes in *controller* are allowed to use services of *model* and *fetcher* (lines 2-3). After computing the DSM, LDM highlights the existing dependency from *view* to *model* as potential violation (Figure 2.3d), since module *view* is not explicitly allowed to access *model* according to the proposed design rules. It is important to notice, however, that the LDM tool does not provide means to detect absences.

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 [Terra and Valente, 2009]. DCL provides constraints to capture divergences and absences. First, to capture divergences, architects specify `only can`, `can only` or `cannot` rules for specified modules. Last, to capture absences, architects specify dependencies that `must` be present in the source code. Moreover, DCL provides software architects with a fine-grained model for the specification of structural dependencies in object-oriented systems, which can be generic (depend) or more specific (e.g., access, declare, create, extend, etc.). `ArchRuby`—the architecture conformance checking technique proposed in this dissertation—is directly inspired on DCL constraints.

In order to check architectural conformance, the architect has to follow three steps. First, it is necessary to specify the system modules. Second, he maps the source code elements to the previous defined modules. Basically, the architect specifies which module represents each part of the source code. Lastly, he defines the constraints among modules, which are continuously enforced by the `dclcheck` tool. Assuming again the aforementioned MVC architecture (Figure 2.1a), the architect should specify the following constraints:

```
1 module model: tweetlist.model.*
2 module controller: tweetlist.controller.*
3 module view: tweetlist.view.*
4 module fetcher: tweetlist.fetcher.*
5
6 only controller can-depend model
7 view must-depend controller
```

First, he defines the modules. Modules *model*, *controller*, *view*, and *fetcher* include, respectively, all classes from packages `tweetlist.model`, `tweetlist.controller`, `tweetlist.view`, and `tweetlist.fetcher` (lines 1-4). On one hand, to capture divergences, a constraint states that only *controller* can establish dependencies with *model* (line 6). On the other hand, to capture absences, another constraint requires that classes from *view* must depend on *controller* (line 7).

Source Code Query Languages (SCQLs): De More et al. describe a general purpose Source Code Query Language (SQCL), called .QL. By general purpose, we mean that is possible to search bugs, detect refactoring opportunities, compute software metrics, and search for code conventions. Moreover, the .QL language has a syntax similar to Structured Query Language (SQL), which facilitate its adoption by the developers. In the context of architectural conformance, it is possible to use .QL to detect source code that does not follow the planned architecture of the system. For example, De Schutter successfully used .QL to perform architectural conformance checks in Certipost—a Belgian electronic communication company [De Schutter, 2012].

As an example, we use again the `TweetList` system to describe a .QL query that checks if a module different than `controller` is using the `fetcher` module:

```
 1 from RefType type_x, RefType fetcher
 2   where
 3     type_x.fromSource()
 4     and not (type_x.getPackage().getName()
 5     .matches("tweetlist.controller"))
 6     and fetcher.getPackage().getName()
 7     .matches("tweetlist.fetcher")
 8     and depends (type_x, fetcher)
 9 select type_x as Type,
10   "Architectural violation :" + type_x.getQualifiedName()
11   "uses" + fetcher.getQualifiedName() as Violation
```

In .QL, `RefType` represents the types that exist in the system. `RefType` also provides functions that can be used such as `fromSource` (that checks if a type is part of the current source code) and `getPackage` (that returns the package name where the type is defined). Therefore, the query checks whether there is a type defined in the system that is not part of the `controller` module (lines 3-5) and that depends on a type of module `fetcher` (lines 6-8). The query outputs the type that is accessing the `fetcher` module with a description indicating the architectural violation (lines 9-11).

Design Test: Brunet et al. [Brunet et al., 2011] proposed an architecture conformance approach using automated tests. Basically, the design tests are test-like programs that—similar to automated software tests—checks if the implementation respects the architectural rules. Moreover, architectural rules are implemented in the same target programming language of the system as a test. Brunet et al. [Brunet et al., 2011] also argue that developers do not need to learn a new programming language or a new technique to perform architectural conformance, which brings more advantages in adopting design tests.

The authors developed a tool, called Design Wizard, that analyzes the structure of the code and uses jUnit to check assertions. Listing 2.2 illustrates an example of a design test in the `TweetList` system.

```
 1 public class FetcherTest extends TestCase {
 2   public void testFetcherCall() {
 3     DesignWizard dw = new DesignWizard("tweetlist.jar");
 4     PackageNode ctr = dw.getPackage("tweetlist.controller");
 5     PackageNode fet = dw.getPackage("tweetlist.fetcher");
 6     Set<ClassNode> callers = null;
 7     for (ClassNode clazz : fet.getAllClasses()) {
 8       callers = clazz.getCallers();
 9         for (ClassNode caller : callers) {
10           assertTrue(caller.getPackage().equals(ctr) ||
11           caller.getPackage().equals(fet));
12         }
13     }
14   }
15 }
```

Listing 2.2: Design test example

This test verifies if classes that are depending on classes from module `fetcher` belongs to module `fetcher` or `controller`. Lines 4-5 get the `controller` and `fetcher` packages, respectively. The loop on lines 7-13 iterates over all classes and then on all callers of the `fetcher` package and checks whether they belong to the `controller` or `fetcher` modules (lines 10-11).

ArchLint: This approach is an architectural conformance process based on static and historical source code analysis [Maffort et al., 2013, 2016]. Basically, `ArchLint` requires two inputs: (i) a high-level specification of the system and (ii) the history of revisions. In `ArchLint`, classes are statically organized in packages (in systems implemented in Java) and packages are logically grouped into components. In this way, components include information on their names and a mapping to source code, using regular expressions. The history of revisions is used to mine dependencies that will be later classified whether they are violations or not based on heuristics.

The authors conducted a study where it was possible to detect absences and divergences with a precision greater than 50%. Moreover, another positive argument that the authors provide is that the technique does not require a very detailed definition of the system architecture, which reduces the dependency on system architects.

2.3 Dynamic Languages

Basically, at its simplest level, the definition of dynamic languages is related to when the compiler type checks the program. Dynamic languages only type checks the program during run time. Differently, static languages enforce type constraints during compile type, i.e., without executing the program [Agesen et al., 1995; Agesen and Holzle, 1995a]. It is worth noting that this definition is different than being typeless. Both statically and dynamically typed languages are typed, although the major difference is when types are enforced. We separate the discussion about dynamically typed languages in two categories. First, we talk about their history and some features bring by the language designers during time. Second, we focus on their features and show some code examples.

History: Lisp is the first dynamically typed language. It is also considered the second-oldest high-level programming language. Lisp syntax is very simple, based on parenthesized prefix notation [McCarthy, 1960]. In this context, Lisp was also pioneer in trying to solve performance issues in its implementation [Gabriel, 1986]. The concept of garbage collector [Jones and Lins, 1996], where the interpreter must handle the allocation and deallocation of memory automatically, was first implemented in Lisp.

The dynamically typed languages have continuously been growing in functionality. Scheme was the first dynamic language to introduce closures, a way to share lexical scoping with function calls [Sussman and Guy L. Steele, 1998]. Basically, functions are treated as first class citizens and can be returned as a value of a function invocation. Thereby, it is possible to define an enclosing function that has access to the lexical scope of the outer function. This functionality helps and simplifies the implementation of many programming tasks, such as those related with Graphical User Interface (GUI) programming.

Smalltalk is another dynamic language that implements object-oriented programming techniques. In Smalltalk, everything is an object, even basic types such as integer and strings. Smalltalk has also extensive meta-programming abilities, which means that is easy to write programs that modify or write other programs [Goldberg and Robson, 1989]. This brings great flexibility to the programmer but, on the other hand, can introduce bugs that are complex to find.

Continuing the evolution of dynamically typed languages, Erlang has emerged to solve problems related to distributed systems. By avoiding static types, Erlang can focus and offer high-level features such as continuous operation [Virding et al., 1996]. Such feature allows code to be replaced in a running system, while the old code is being running during the same time. For instance, telephone exchanges, and air traffic control systems benefit from this functionality [Virding et al., 1996]. Moreover, the language also includes features to detect run-time errors to guarantee robustness.

Perl was originally proposed as a general purpose scripting language to be used as a glue for Unix. The language facilitates the work with files, text-processing, database manipulation, network management, etc. [Schwartz, 2011]. To this purpose, Perl does not impose arbitrary limitations on program data. For example, strings and arrays can grow as large as they need to (as long as the system has available memory for it). Perl also supports closures, class-based method dispatch (i.e., polymorphic operations), lexical scoped variables, etc. The Perl interpreter also makes legal type conversions automatically, for example, conversions from number to string [Schwartz, 2011].

With the evolution of the World Wide Web (WWW), PHP emerges to fill the gap between server-side scripting and Hypertext Markup Language (HTML) [Converse, 2002]. PHP is an interpreted language that offers reflexion features to the programmers. For example, it is possible to inspect programs at run time and even execute arbitrary PHP code with the `eval` function call. This flexibility helps in the creation of frameworks that facilitate the development for programmers. For instance, Laravel is a framework that uses reflexion to automatically inject the dependencies required by a software system [Bean, 2015].

JavaScript is another dynamically typed language whose primary focus is on client-side web development. The language does not force type constraints, therefore situations where the interpreter converts values from one type to another are usual. For instance, if a programmer tries to sum a number with a string, the interpreter will convert automatically the number to string and concatenate them [Flanagan, 2006]. Moreover, it is possible to use JavaScript in the server-side through Node.js, which provides bindings to low-level Unix APIs for working with processes, files, network sockets, etc.

Nowadays, Ruby and Python are popular dynamic languages among developers [TIOBE index, 2017]. As they are interpreted, they can be easily ported to different platforms [Thomas et al., 2004]. Ruby and Python also provide extensive standard library bundled with too many features, which contributes to the spread of these languages. Moreover, they are largely used in the web development context due to popular frameworks such as Ruby on Rails (for Ruby) and Django (for Python).

Features: The flexibility offered by dynamically typed languages helps in the development effort needed to develop software systems [Palsberg and Schwartzbach, 1991]. For example, dynamically typed languages allow developers to change the program structure at run time. In other words, developers have freedom to modify or add classes, methods, and modules at run time. In this context, next we briefly introduce some key features offered by dynamically typed languages.

First, dynamically typed languages provide a richer set of data types than statically ones [Thomas et al., 2004; Ascher and Lutz, 1999]. Lists (arrays that can grow as long as the system has enough memory) and strings (that can also grow arbitrarily) are the two most common among them. Furthermore, dynamically typed languages also provide support for associative arrays or hash tables (a fast key/value lookup structure), and sets. The syntax varies from language to language but they are frequently used in libraries and frameworks [Greenfeld and Greenfeld, 2015; Hartl, 2012]. The following code illustrates an example of hash in Python:

```
1 hash = {}
2 hash["first_key"] = "first string"
3 hash["second_key"] = "second string"
4 hash
5 > {"first_key": "first string",
6 "second_key": "second string"}
```

In line 1, we define a variable called **hash** and assign an empty hash to it. In lines 2-3, we create two keys with two strings as values. Finally, in line 4 we ask the interpreter to evaluate the **hash** variable and the result is presented in lines 5-6. We can keep creating hash keys as long as we have enough memory for them.

Metaprogramming is another feature that is common in dynamically typed languages. It came from Lisp (and Smalltak) [Perrotta, 2010] and refers to the ability of a program to query, manipulate, or create a program by itself. Basically, it means that a program can read, generate, analyze, or transform itself at run time [Czarnecki and Eisenecker, 2000]. For example, Ruby can perform deep querying in objects to discover information about types, how many parameters are expected by a method, or even create methods or modify existing ones [Perrotta, 2010]. The following code shows an example of Ruby code that uses this technique:

```
def add_attribute(klas, attribute)
  klas.class_eval do
    define_method "#{attribute}=" do |value|
      instance_variable_set("@#{attribute}", value)
    end
  end
end
add_attribute(Professor, :name)
```

Line 1 defines a method called `add_attribute` that receives two arguments. Lines 2-6 define a new method, using as name the value of the `attribute` parameter. This new method sets a new instance variable to the class received through the `klas` parameter. Lastly, line 8 calls the `add_attribute` method to add the `name` method in the `Professor` class.

Eval feature is the ability to evaluate arbitrary code expressions represented as strings [Perrotta, 2010]. For example, it is possible to read a file or create a string containing valid code, and execute that code at run time. When carefully used, this feature can help developers with configuration that is needed to perform in the system [Black, 2009]. For example, it is possible to execute code provided by users of the system. However, eval has its downsides too. The main one are related to security. A malicious user can craft a program to perform unauthorized tasks, as illustrated in the following example:

```
1 params = "exec('cat /etc/passwd')"
2 eval("#{params}")
```

In line 1, we define a string containing a valid Ruby code (suppose this string came from an user input, for example) and, in line 2, we use eval to evaluate that string at run time. In that case, if the Ruby process has permission to read the `passwd` file, the content of this file will be shown.

Closure is the ability offered by dynamically typed languages to retain access to variables from the scope that a function is defined [Perrotta, 2010]. For example, closure is a defining feature of JavaScript [Resig and Bibeault, 2013]. More specifically, as defined by Resig and Bibeault [Resig and Bibeault, 2013], closure is the scope created when a function is declared, and such scope allows the function to access and manipulate variables that are external to it. The following JavaScript code illustrates the use of closures:

```
var functionReference;
function test(){
  var aVariable = "inside test function";
  function insideTest(){
    console.log(aVariable);
  }
  functionReference = insideTest;
}
test();
functionReference();
```

Basically, in line 1, we define a variable called `functionReference`. Then, we define a function called `test` (lines 2-8) that declares a local variable named `aVariable` (line 3), a new function called `insideTest` that outputs the value of `aVariable` on the console (line 5), and assigns a reference to function `insideTest` to variable `functionReference` (line 7). In line 9, we invoke `test`, which causes the declaration of the `insideTest` function and assign its reference to variable `functionReference`. Finally, in line 10, we invoke `insideTest` through the `functionReference` variable, which causes the value of "inside test function" to be printed on the console.

The flexibility offered by dynamically typed languages, previously discussed in this section, tends to be useful during development, once the programmers are not constrained by rules imposed by type systems. On the other hand, it impacts performance and postpones type checking to runtime [Agesen et al., 1995]. Performance is affected due to run-time checking and look-up processes that are extensively used by dynamically typed languages.

2.4 Ruby

Since our approach focuses on systems implemented in Ruby, an overview of this language is relevant. Ruby is a dynamically typed language, which is interpreted and is purely object oriented. Even *true* and *false* are objects, i.e., they are instances of `TrueClass` and `FalseClass`, respectively [Black, 2009]. This brings a great expressiveness for the language since it is natural to read messages sent to objects. For example, consider the problem of converting a string to integer. To perform that in Java we need to do something like the following code:

```
int foo = Integer.parseInt("1234");
```

However, in Ruby a string "knows" how to convert itself into an integer. In other words, the ability to convert a string into an integer is built into the String class of the language. Therefore, to accomplish the conversion we just need to send the message `to_i` to a string, as in the following code:

```
int_value = "1234".to_i
```

Ruby syntax aims to make programs easy to read. For example, there is no need to end statements with semicolons as long as each one is located in separated lines. Moreover, it is also possible to omit parentheses when defining or calling methods [Thomas et al., 2004], which removes extra characters. Suppose we have an object that responds to a `name` method and it is stored in a variable called `person`. The following code exemplifies that:

```
name = person.name
```

Identifiers in Ruby can be variables, constants, keywords, and method names. First, variables can be split into local, instance, class, and global variables. Local variables start with a lowercase letter or an underscore, and can be completed with letters, underscores, and/or digits. For instance, `_x`, `user_name`, `className`, `__userData__` are all valid local variable names. It is important to notice that the Ruby community has a convention to use underscores rather than CamelCase when composing local variable names with more than one word. Instance variables start with an @ sign and are used to store information for individual objects. Class variables start with two @ signs and store information per class hierarchy. Global variables starts with a $ sign and can be accessed anywhere in the program [Black, 2009].

Constants begin with an upper case letter. For example, `String` is an valid constant name in Ruby. On the other hand, keywords are mostly composed by lowercase letters and single-words. There are, approximately, 40 reserved words in Ruby. For instance, `def` (for method definitions), `class` (for class definitions), `if` (conditional execution), and `while` (repetition) are reserved words in Ruby. For method names, the same rules and conventions of local variables are applicable and they can use the symbols ?, !, or = to end their names to make their meaning more expressive [Thomas et al., 2004].

Arrays and hashes are indexed collections in Ruby and they can grow as needed to handle new elements as long as the system has memory [Thomas et al., 2004]. Regarding the keys, arrays are indexed by integers and hashes support any object as key. Moreover, as being a dynamically typed language, any array or hash can hold different types of objects [Thomas et al., 2004].

In Ruby, all data structures and values are treated as objects, as expected in a purely object-oriented language. Therefore, every object is capable of understanding a set of messages, which are defined by the object class. Methods can be called from the class they are defined, from other classes, or from the enclosing classes (depending on accessibility rules) [Black, 2009]. Messages are sent by using the dot operator: the receiver is located on the left and the message on the right.

Furthermore, Ruby is a dynamic language with several powerful abstractions [Black, 2009]. For example, it is possible to re-open a class, evaluate a valid Ruby code inside some context (eval), re-define methods, and call methods passing their name as strings. Metaprogramming is used to manipulate these language abstractions to modify the code during system execution [Perrotta, 2010]. However, it is necessary to be careful when modifying the language core since the changes are global and will take place as long as the program starts to run [Black, 2009].

Although Ruby has single inheritance, it is possible to include modules in one class. A module implements methods and constants, but unlike a class, it is not possible to instantiate a module [Thomas et al., 2004]. Therefore, a module is a way to collect and encapsulate behavior that can be shared among objects. Developers need to be careful about the modules that are included into a class. For example, if two different modules define methods with the same name the Ruby interpreter executes the first that is returned during the method lookup process [Black, 2009].

As an example, Listing 2.3 illustrates the usage of the aforementioned features. In line 1, the code defines class `RbClass` and a module `Test`. In line 2, on the right, we define a method called `salute` that receives an argument named `language`. Also in the `salute` method, we define a hash with two keys (line 3). In line 2, on the left, class `RbClass` includes module `Test`; therefore method `salute` is now part of this class. In line 8, we instantiate an object of `RbClass` and call method `say_hi` with the parameter `:en` (line 9). In line 11, we re-open class `RbClass` and define a new method called `say_bye` (lines 12-14). In lines 17-19, we define an `add` method only for the object `o`, which is invoked in line 21.

```
 1 class RbClass    module Test
 2   include Test     def salute(language)
 3                      terms = {:pt_br => "Ola",:en => "Hi"}
 4   def say_hi           puts terms[language]
 5     salute         end
 6   end            end
 7 end
 8 o = RbClass.new
 9 o.say_hi(:en)
10
11 class RbClass
12   def say_bye
13     puts "bye"
14   end
15 end
16
17 def o.add(x,y)
18   x.send "+", y
19 end
20
21 o.add(5,9)
```

Listing 2.3: Ruby source code example

2.5 Type Inference

As explained in Section 2.3, dynamically typed languages do not check types during compile time. However, type checking has many benefits, such as providing more legible and documented code, facilitating the implementation of code analysis tools, more reliable refactoring activity, promoting the construction of better IDEs (i.e., with auto-complete support), and achieving earlier error detection [Palsberg and Schwartzbach, 1991]. Therefore, a type inference algorithm can be used together with dynamically typed language to provide the benefits associated with type checking.

Before discussing type inference algorithms, is important to define what is a type. Type generally refers to several distinct concepts, such as abstract types in Java (interfaces), concrete type (implementations), or set of classes [Agesen and Holzle, 1995a]. In this dissertation, we use type as a synonym for class, which is the structure used in the object-oriented paradigm to describe the implementation of object instances. Therefore, the type of a variable is the set of possible classes that can be assigned to it.

In order to explain the problems that dynamic typing can bring to developers, we use the following Ruby code:

```
1 class Professor
2   def assign_to_class(class)
3     class_name = class.name
4     classes.add(class_name)
5   end
6 end
```

The `Professor` class has a method named `assign_to_class` that receives one argument (line 2). However, by just reading the code a developer cannot infer what is the type expected to be received in the `assign_to_class` and cannot tell what is the type of `classes` variable (line 4). This small snippet of code can go wrong during the system execution. For example, suppose that an instance of `Professor` receives a method call that is mapped to `assign_to_class` method, passing an object as parameter that does not define a `name` method. Moreover, suppose that nothing was assigned to the `classes` variable previously. These problems can only be discovered at run time.

The goal of type inference is to discover the types a variable can assume [Palsberg and Schwartzbach, 1991; Agesen et al., 1995; Agesen and Holzle, 1995a]. This process can be done statically (i.e., without executing the system) or dynamically (i.e., executing the system and analyzing it at run time). When performed statically, type inference relies on heuristics to discover the types an expression can produce during system execution.

The following code illustrates a snippet of code, from an open-source system called Vagrant, which is complex to infer types with a static algorithm:

```
1 def self.server_url(config_server_url=nil)
2   result = ENV["VAGRANT_SERVER_URL"]
3   result = config_server_url if result=="" or result==nil
4   result || DEFAULT_SERVER_URL
5 end
```

First, the local variable `result` receives the value stored in the `ENV` hash (line 2). Then, the value of `result` can change to the return value of the `config_server_url` method if the `result` variable contains an empty string or a `nil` value (line 3). To infer correctly the type of the `result` a static algorithm should know the values stored in all keys of the `ENV` hash. Moreover, it should know how to analyze the conditional structure (line 3) since an `if` expression can change the value stored in `result`.

However, there are simple cases when a static algorithm can easily infer the type of a variable. The following snippet of code illustrates one of such cases:

```
1 class Professor
2   def name
3      "John Doe"
4   end
5 end
6 p = Professor.new
```

In lines 1-5, a `Professor` class is defined with one method called `name` (lines 2-4). In line 6, an instance of `Professor` is assigned to the `p` variable. Therefore, the type of this variable is `Professor`. A static algorithm can easily infer this by just analyzing the Abstract Syntax Tree generated for the program.

On the other hand, due to complexity of dynamically typed languages, static algorithm for type inference is complex to be developed. Johnson [Johnson, 1986] suggested some algorithms to Smalltalk systems. They implemented an example where local variables are considered to have the same type in the entire method. Specifically for Ruby, Furr et al. [Furr et al., 2009] proposed an approach that requires type annotations when it cannot infer types automatically.

To help static approaches, we can use dynamic ones [Agesen and Holzle, 1995a; Johnson, 1986]. In this case, the system is analyzed at run time. Obviously, the program needs to run with a profiler that collects data generated during its execution [Agesen and Holzle, 1995a]. The following code, developed in Ruby, illustrates a situation where a dynamic approach can successfully infer a variable type.

```
1 eval("
2   class Options
3     def values_for_operation
4       PermittedValues.new(["exp", "def", "mod"])
5     end
6 ")
7
8 opt = Options.new
9 permitted_values = opt.values_for_operation
```

First, the program uses the `eval` function to evaluate a valid Ruby code during the system execution (lines 1-6). Specifically, the code passed to `eval` creates a class named `Options` (line 2) with a method named `values_for_operation` (line 3) that returns a new instance of a `PermittedValues` class (line 4). Next, in line 8, an instance of class `Options` is instantiated and assigned to the `opt` variable. In line 9, variable `permitted_values` receives the value returned by method `values_for_operation`.

In the previous example, it is important to notice that the type of `permitted_values` is hard to obtain with a static type inference algorithm. In this case, a static approach needs to know how to evaluate strings passed as an argument for the `eval` function. On the other hand, a dynamic algorithm that profiles the system during execution has an advantage to access information that is generated by the language virtual machine itself.

As a consequence of type inference, both security in messages sent to objects and run-time performance can be increased in dynamically typed languages [Palsberg and Schwartzbach, 1991]. For security, wrong method invocation tends to not occur since type inference may discover which type an expression can have. For example, suppose class `A` does not implement method `foo` and variable `b` is an instance of `A`. In this sense, it is not possible to call method `foo` from variable `b`. This verification is possible because the information about the type permit to verify if the receiver of the message implements a method to handle the message [Palsberg and Schwartzbach, 1991].

Additionally, for performance, the interpreter can use the inferred type information to inline message calls at run-time, eliminating the need of searching for the receiver [Agesen and Holzle, 1995b]. For example, if the interpreter knows that a particular receiver is of type `Professor` and that the message sent is defined in the

`Professor` class, it can inline this message. This removes the need of searching for the receiver and for the particular fragment that implements the code to execute the message.

2.6 Final Remarks

This chapter provided necessary background to understand the architectural conformance solution proposed in this dissertation. In Section 2.1, we introduce the idea of architectural conformance and outlined some concepts related to it. We also described the importance of architecture to the software evolution.

In Section 2.2, we presented different architectural conformance approaches, which are related to our work in terms of violation detection and architectural visualization. Moreover, our approach is directly inspired by DCL, Reflexion Models, and DSM.

In Section 2.3, we described some dynamically typed languages, once our approach for architectural conformance aims to support developers that use these language. We also described some central features of these languages. To complement, in Section 2.4, we detailed some features related to Ruby, which is the language we chose to implement our approach.

Finally, in Section 2.5, we detailed what is type inference since we focus on dynamically typed languages that check type at run time. Therefore, type inference algorithms help static architectural conformance techniques performed in dynamically typed languages. In this context, we presented advantages that type inference algorithms bring to dynamically typed languages. Moreover, we illustrated with code examples the difficulty imposed by dynamically typed languages on the type inference process when performed with static techniques.

In the next chapter, we present our architectural conformance and visualization approach, including a running example with full details about how to specify architectural rules, how to perform architectural conformance, and how to produce high-level visualizations of the concrete architecture. We also detail our heuristic to perform type inference.

Chapter 3

Proposed Approach

Chapter 2 provided the background necessary to understand this master dissertation. In this chapter, we describe the approach proposed in this master dissertation: an architectural conformance technique based on static code analysis and on a lightweight type inference heuristic targeting systems implemented in dynamically typed languages. Additionally, we also propose two high-level architectural models to better visualize architectural violations.

This chapter is organized as follows. Section 3.1 provides an overview of the proposed approach. Section 3.2 presents the running system. Section 3.3 details the specification of architectural rules and Section 3.4 describes the proposed architectural conformance process. Section 3.5 presents the high-level architectural models our approach relies on to better visualize the identified architectural violations. Section 3.6 presents the type inference heuristic used in our work. Section 3.7 presents a prototype tool that implements our approach. Finally, Section 3.8 concludes with a general discussion.

3.1 Overview

As described in Chapter 2, several architectural conformance techniques have already been proposed to detect architectural violations. However, none of them addresses the particularities imposed by dynamically typed languages. Therefore, our central goal in this master dissertation is to provide developers with means to control the architectural erosion process in systems implemented in dynamically typed languages, by reporting architectural violations (architectural conformance) and by providing high-level architectural models to better visualize the identified violations (visualization).

Figure 3.1 retakes an overview of 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 distinguish the dependencies—edges in reflexion models and cells in DSMs—that represent violations (refer to Chapter 2).

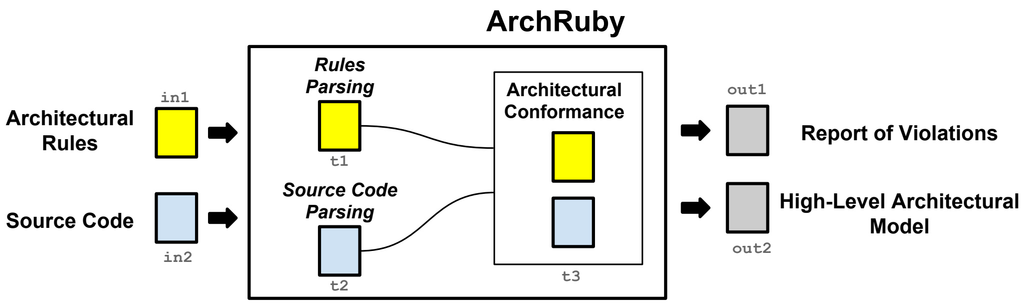


Figure 3.1: The proposed approach

3.2 Running Example

We rely on the architecture of `ArchRuby`[1] itself and its implementation to illustrate the architectural conformance and visualization processes provided by our approach. The tool is implemented in Ruby and relies on five Gems:[2] `RubyParser` to parse the source code, `SexpProcessor` to perform tree traversals, `Yaml` to parse the architectural rules specification file, `GraphViz` to produce reflexion models, and `IMGKit` to produce DSMs. Figure 3.2 shows the diagram of the core classes of the system. A more detailed description on the `ArchRuby` implementation can be found in Section 3.7.

3.3 Architectural Rules Specification

Architectural rules are specified in a domain-specific language in YAML format, which is a format widely used in the Ruby ecosystem. Thereupon, even non-experienced

[1]The source code is publicly available at `http://github.com/sergiotp/archruby`.
[2]Gem represents a reusable package or application written in Ruby language.

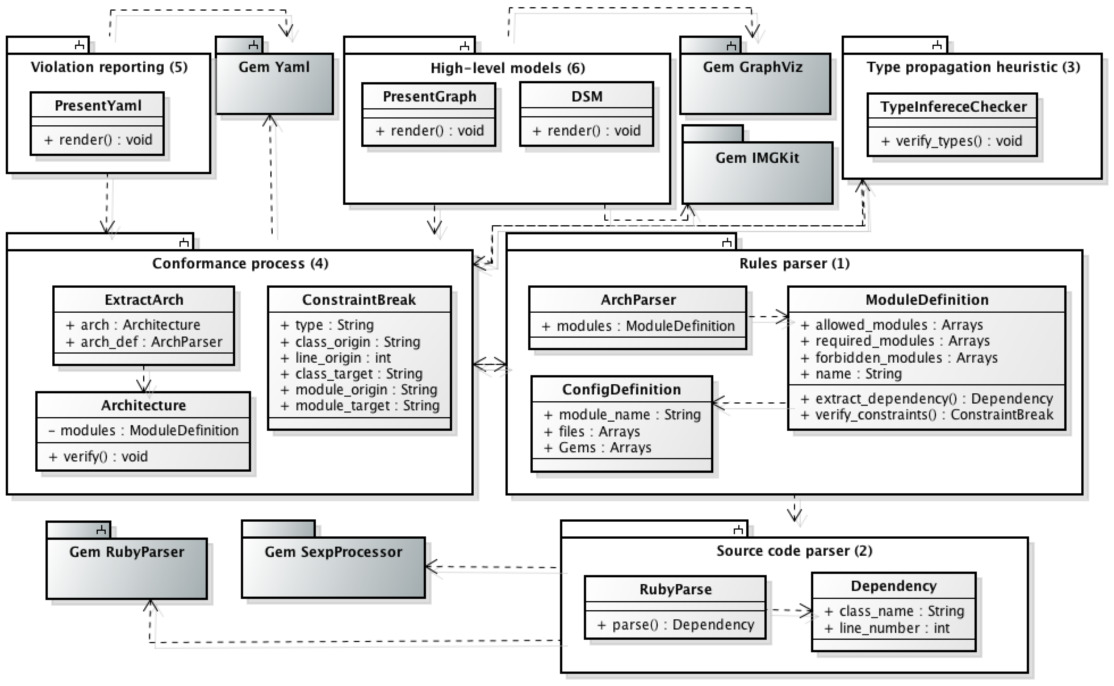


Figure 3.2: `ArchRuby` architecture

developers can easily define rules. Specifically, each module of the system under evaluation must be formalized as follows:[3]

```
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>*, 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*[4] (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 [Passos et al., 2010]—for each module we define

[3]Formalization based on the Extended Backus-Naur Form (EBNF).

[4]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`

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.[5] Likewise, to detect absences—dependencies that do not exist in the source code but are required by the planned architecture [Passos et al., 2010]—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, Listing 3.1 presents the definition of architectural rules to the `ArchRuby` tool. For example, module `module_definition` (lines 1-4) 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 6-8) 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 15) 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).

[5]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 $M3$ 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.

```
module_definition:
  files: 'lib/archruby/architecture/module_definition.rb'
  allowed: 'config_definition, ruby_parser, dependency,
  constraint_break, file_extractor'

multiple_constraints_validator:
  files: 'lib/archruby.rb'
  forbidden: 'architecture'

architecture_parser:
  files: 'lib/archruby/architecture/parser.rb'
  allowed: 'config_definition, module_definition,
  type_propagation, yaml_parser'

presenters:
  files: 'lib/archruby/presenters/**/*.rb'
  allowed: 'architecture, graphviz, imgkit'

ruby_parser:
  files: 'lib/archruby/ruby/parser.rb'
  allowed: 'dependency'
  required: 'parser_ruby, sexp_processor'

config_definition:
  files: 'lib/archruby/architecture/config_definition.rb'

architecture:
  files: 'lib/archruby/architecture/architecture.rb'
  forbidden: 'type_propagation'

constraint_break:
  files: 'lib/archruby/architecture/constraint_break.rb'

dependency:
  files: 'lib/archruby/architecture/dependency.rb'

type_propagation:
  files: 'lib/archruby/architecture/type_propagation_checker.rb'

file_extractor:
  files: 'lib/archruby/architecture/file_content.rb'

parser_ruby:
  gems: 'RubyParser'

sexp_processor:
  gems: 'SexpInterpreter'

yaml_parser:
  gems: 'YAML'

graphviz:
  gems: 'GraphViz'

imgkit:
  gems: 'IMGKit'
```

Listing 3.1: `ArchRuby` architectural specification file

3.4 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 inference heuristic (it is described in Section 3.6); 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` (Listing 3.1). In such specification, module `module_definition` is not explicitly allowed to depend on module `type_propagation` (lines 3-4). However, we intentionally made a class from `module_definition` to access a class from `type_propagation`. Such dependency represents a violation and is reported to developers in the textual output file as illustrated in Figure 3.3.[6] 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 rule that forbids such dependency (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 next in Section 3.5.

```
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_definition' cannot depend on module 'type_propagation'
```

Figure 3.3: Textual report of an architectural violation

3.5 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. [Murphy et al., 1995] and (ii) Dependency Structure Matrix (DSM) in a subtle adaptation of the one proposed by Sangal et al. [Sangal et al., 2005].

[6]The report is also in YAML format to facilitate reuse.

3.5.1 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 2.1).

Figure 3.4 illustrates the reflexion model of `ArchRuby`. 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):

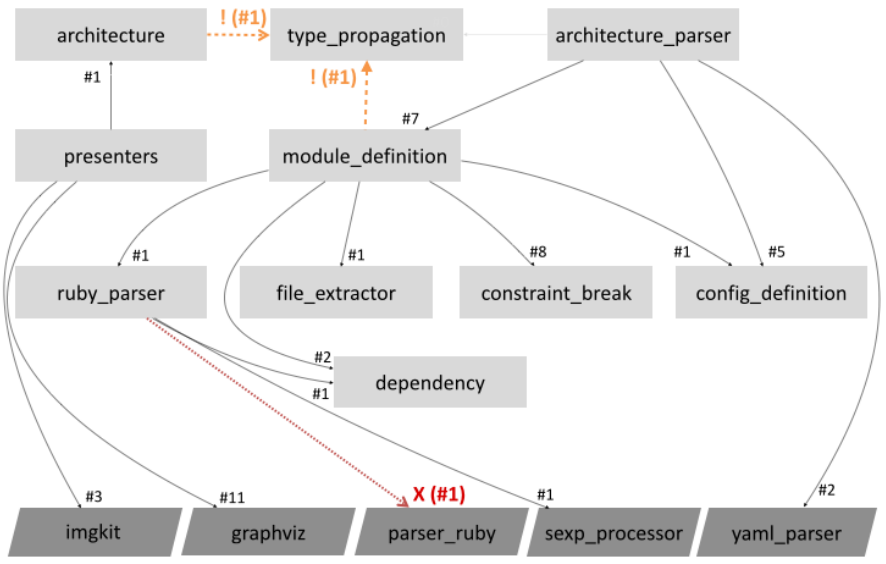


Figure 3.4: Reflexion model automatically computed by `ArchRuby`

(→) 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, Listing 3.1).

($\overset{!}{\dashrightarrow}$) 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, Listing 3.1). As another example, `module_definition` depends on module `type_propagation`, but it is not explicitly allowed (case *ii*; see line 3, Listing 3.1).

($\overset{\times}{\dashrightarrow}$) 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, Listing 3.1).

($\rightarrow$) 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, Listing 3.1), but there is no dependency from the former to the latter.

3.5.2 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 2.1).

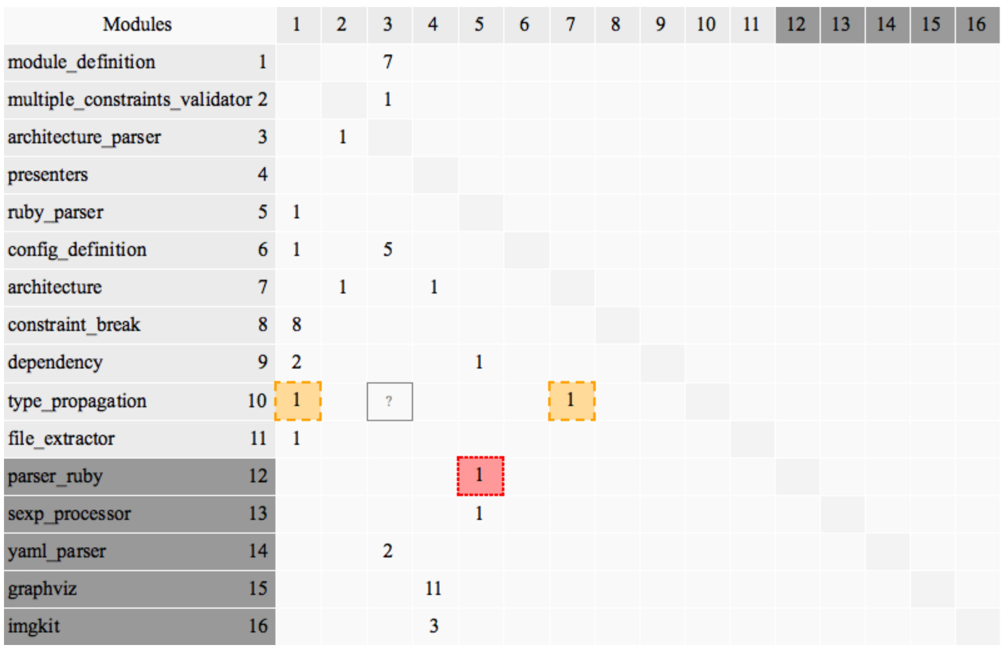
Figure 3.5 illustrates the DSM of `ArchRuby`. The cells represent the number of references between two modules. The cells are shown as follows:

(⬜) Gray cell: indicates an *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.

($\boxed{?}$) 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 3.5: DSM automatically computed by `ArchRuby`

3.6 The Proposed Type Inference Heuristic

In this section, we describe a type inference heuristic—more specifically, a simplification of the one formalized by Furr et al. [Furr et al., 2009]—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`] ∈ `TYPES`.

ii) **Recursive step**: If [`f`, `x`, `T`] ∈ `TYPES` and there is a call `g`(`x`) in `f`, then [`g`, `y`, `T`] ∈ `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 `TYPES`.

Listing 3.2 illustrates the proposed heuristic. When executing the base step of the algorithm, it initializes `TYPES` with [`A`::`f`, `x`, `Foo`], [`A`::`f`, `b`, `B`], [`A`::`f`, `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`, `Foo`] and [`B`::`g`, `z`, `A`] are included in `TYPES`, since the type of the variables `x` and `self` are known in the call of `g`. On the second application of the recursive step, the triples [`C`::`h`, `y`, `Foo`] and [`C`::`h`, `y`, `A`] are included in `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`, `Foo`] and [D::m, k, A] are included in 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 forth application of the recursive step reaches the fixpoint since no new triple is added to set TYPES.

```
class A                    class B                class C
  def f                      def g(x z)             def h(y)
    x = Foo.new               c = C.new               d = D.new
    b = B.new                 c.h(x)                  d.m(y)
    b.g(x,self)               c.h(z)                end
  end                        end                  end
end                        end
```

Listing 3.2: 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 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 Foo, in this scenario) when searching for violations.

3.7 The ArchRuby Tool

ArchRuby is a Gem for Ruby that implements our proposed approach [Miranda et al., 2015a]. The tool is executed from the command line. We decided for such User Interface (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 3.1.

As also previously illustrated in Figure 3.2, 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 3.6. 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 3.4. 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 3.2, 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 3.5. 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, it 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 tool. 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`.

3.8 Final Remarks

Software systems evolve to respond to users demands. As a consequence—due to lack of knowledge, short deadlines, etc.—the architectural patterns tend to deteriorate and hence nullifying the benefits provided by an architectural design, such as maintainability, scalability, portability, etc. Due to its relevance, several techniques have been proposed to tackle this problem. However, none of them addresses the particularities that dynamically typed languages require to verify the system architecture.

Therefore, in this chapter, we proposed an approach that provides ways to specify the system architecture, to perform architectural conformance process, and to verify architectural violations through a textual report or by means of two high-level visualization models. Moreover, we proposed a type inference heuristic that increases the number of dependencies analyzed during conformance process. Finally, we designed a prototype tool, called `ArchRuby`, that implements our approach.

In the next chapter, we conduct an evaluation of our approach in real contexts of software development. We rely on the architects of three real-world systems to specify the system architecture and to validate the architectural conformance processes performed by our approach.

Chapter 4

Proposed Approach Evaluation

This chapter evaluates the applicability of our proposed approach. 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.

We organized this chapter as follows. Section 4.1 describes the target systems. Section 4.2 details the methodology we used to conduct the evaluation. Sections 4.3, 4.4, and 4.5 report the results for `Dito Social`, `Tim Beta`, and `PLC Attorneys`, respectively. Section 4.6 presents a general discussion of our results. Section 4.7 describes threats to validity. Finally, Section 4.8 concludes this chapter.

4.1 Target Systems

We evaluate our solution in three real-world systems: `Dito Social`[1], a social platform provided by an IT company to its final customers; `Tim Beta`[2], a telecommunication company communication channel with mostly target young groups; and `PLC Attorneys`[3], a project task management software system used by a law firm. Table 4.1 reports the main information of the systems.

[1]`http://www.dito.com.br`
[2]`http://www.timbeta.com.br`
[3]`http://metodo.plcadvogados.com.br`

Table 4.1: Target systems

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

4.2 Methodology

For each subject system with the support of its chief architect who designed the architecture to be evaluated, 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 3.3). 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 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 resulting 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.

4.3 Dito Social

Architectural rules specification: The software architect specified 62 modules and 43 architectural rules. A relevant subset of the specification is reported in Listing 4.1 (the complete architectural definition is available in Appendix B). 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 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 provide data and e-mail delivery functionality (e.g., `post_model`, `interaction_model`, `mail`, `aws`, etc.) (lines 22-23).

```
 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,
23   mail, aws, http_party'
```

Listing 4.1: Subset of the architectural specification of `Dito Social`

Architectural conformance: `ArchRuby` could detect 24 violations in `Dito Social`, as reported in Table 4.2. Two of these violations are discussed next.

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

Table 4.2: 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

shown in Listing 4.2, `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 Listing 4.1). 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
```

Listing 4.2: 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 Listing 4.3, `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 Listing 4.1). 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).

```
class Post                                  # from Module post_model
  def first_update_complete_info_from_facebook(post_info, update_freq,
    limit = 50, is_customer = false)
    ...
    vpost = Post.new(fb_id: fb_id, likes_count: post_info['likes'],
          updated_info: true, premium: premium, international: international)
    facebook = FacebookInfoRetriever.new
    facebook.get_first_likes_comments_and_people(vpost, limit,
             special_token.present?) do |info|
    ...
  end
end

class FacebookInfoRetriever         # from Module facebook_info_retriever
  def get_first_likes_comments_and_people post, limit = 25,
    special_token = false, &block
    ...
    likes_count = post['likes']
    ...
  end
end
```

Listing 4.3: Example #2 – Divergence detected in `Dito Social` by type propagation

Architectural visualization: Figure 4.1 illustrates a fragment of the reflexion model (the complete reflexion model is available in Appendix B). 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.

4.3.1 Discussion

The software architect described the architectural rules incrementally. According to the architect, this way 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

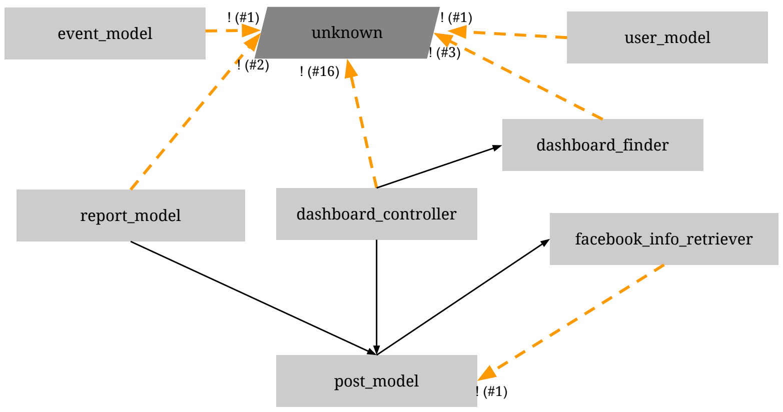


Figure 4.1: Fragment of the reflexion model of `Dito Social`

`facebook_info_retriever` and `post_model`. Module `facebook_info_retriever` is likely to be used only with 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 complementary, 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.

4.4 Tim Beta

Architectural rules specification: The software architect specified 43 modules and seven architectural rules. A relevant subset of the specification is reported in Listing 4.4 (the complete architectural definition is available in Appendix C). Module `models` implements the Model layer of the MVC architectural pattern and hence accesses the modules that are responsible for the data persistence (lines 3–7). Module `core` implements the main features of the system and hence accesses 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'
```

Listing 4.4: Subset of the architectural specification of `Tim Beta`

Architectural conformance: `ArchRuby` could detect 22 violations in `Tim Beta`, as reported in Table 4.3. An example of a detected violation is discussed next.

Table 4.3: Architectural violations detected in `Tim Beta`

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

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 Listing 4.5, 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 Listing 4.4).

```
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
```

Listing 4.5: Example #3 – Divergence detected in `Tim Beta`

Architectural visualization: Figure 4.2 illustrates a fragment of the reflexion model to better visualize some identified violations (the complete reflexion model is available in Appendix C). 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.

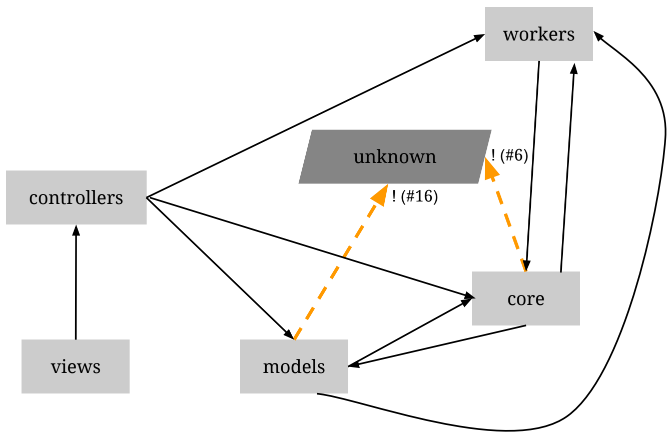


Figure 4.2: Fragment of the reflexion model of `Tim Beta`

4.4.1 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 fragments 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.

4.5 PLC Attorneys

Architectural rules specification: The software architect specified 14 modules and 11 architectural rules. A relevant subset of the specification is presented in Listing 4.6

(the complete architectural definition is available in Appendix D). 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 13). Module `mailers` is responsible for triggering e-mails to clients and can access modules that provide information about the projects (line 17). Finally, module `controller` is responsible for handling users requests and can access modules that contain data it needs (lines 22-24).

```
 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,
 8             app/models/areas_project.rb, app/models/attack.rb,
 9             app/models/control.rb, app/models/diagnostic.rb,
10             app/models/improvement.rb, app/models/action.rb,
11             app/models/task.rb, app/models/responsible.rb'
12   required: 'activerecord'
13   allowed:  'chartdraw, mailers, admins'
14
15 mailers:
16   files:    'app/mailers/**/*.rb'
17   allowed:  'project_relations, project'
18   required: 'actionmailer'
19
20 controller:
21   files: 'app/controllers/**/*.rb'
22   allowed: 'presenters, devise, actioncontroller,
23            project_relations, project, admins,
24            consolidated_control'
```

Listing 4.6: Subset of the architectural specification of `PLC Attorneys`

Architectural conformance: `ArchRuby` could detect two violations in `PLC Attorneys`, as reported in Table 4.4. 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 4.4: 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 Listing 4.7, 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 Listing 4.6). This violation is inevitably critical because the e-mails will not be delivered without the establishment of the dependency with `ActionMailer`.

```ruby
class DiagnosticsMailer #from Module mailers
  default from: "test@test.com"

  def diagnostic_created(admin, project_id, area_id)
    @admin = admin
    @project = Project.find(project_id)
    @area = Area.find(area_id)

    mail(to: @admin.email, subject: '[PLC - Added new diagnostic!')
  end
end
```

Listing 4.7: Example #4 – Absence detected in `PLC Attorneys`

Architectural visualization: Figure 4.3 illustrates a fragment of the reflexion model to better visualize some identified violations (the complete reflexion model is available in Appendix D). 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`.

4.5.1 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 has not established a required dependency between modules `mailers` and `actionmailer`. Without such

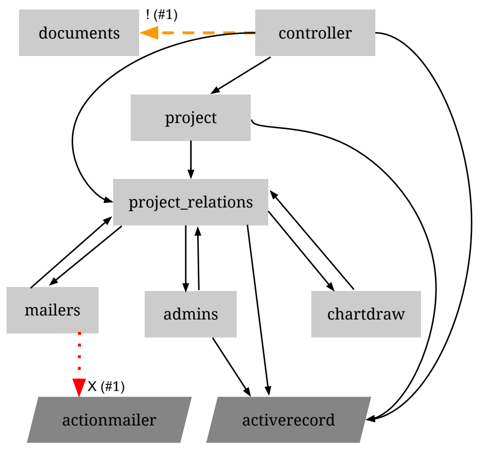


Figure 4.3: Fragment of the reflexion model of `PLC`

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.

4.6 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 indicates that developers establish dependencies with modules that are forbidden (or not explicitly allowed) by the architectural rules; (iii) on the other hand, we could detected 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.

4.7 Threats to Validity

There are two main threats to validity of the study [Wohlin et al., 2012]. 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 being 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.

4.8 Final Remarks

The proposed approach is based on static analysis techniques and on a type propagation heuristic to address particularities of dynamically typed languages. In order to validate this approach, we conducted an evaluation in three real-world software systems implemented in the Ruby language.

We observed that architectural rules specification and architectural conformance are an iterative process. In other words, the architectural rules specification was refined by the architects in order to avoid false positives. Regarding the conformance process itself, `ArchRuby` detected a high number of divergences in the three analyzed system. The software architects had no previous knowledge on these violations and reported that they negatively impact on the maintenance tasks. Some of these violations could only be detected due to our type inference heuristic.

In the next chapter, we evaluate the proposed type inference heuristic. Basically, we perform the type inference heuristic in our previous three real-world systems and in 28 other open-source systems, and analyze the number of dependencies that are detected due to the heuristic. Moreover, we compare the proposed type inference heuristic with dynamic techniques w.r.t. the accuracy. The goal is to investigate if dynamic techniques can complement our approach.

Chapter 5

Type Inference Evaluation

Dynamically typed languages do not enforce types during development. However, this information is important to architectural conformance processes. Therefore, as described in Chapter 3, we implemented a simple heuristic that infers types in dynamically typed languages to improve the number of analyzed dependencies. In this chapter, our goal is to describe a study we conducted to check: (i) whether the proposed type inference heuristic really increases the number of analyzed dependencies, (ii) whether static techniques can be improved by also considering the results of dynamic techniques, and (iii) whether type information provided by dynamic analysis can help in the architectural conformance process.

We organized this chapter as follows. Section 5.1 investigates the effectiveness of the proposed heuristic. Section 5.2 compares our static approach with dynamic one. Section 5.3 measures the impact of dynamic analysis on the PLC system. Finally, Section 5.4 concludes this chapter.

5.1 Effectiveness of the Type Inference Heuristic

The proposed type propagation mechanism, as described in Section 3.6, aims to raise the effectiveness of `ArchRuby` by increasing the number of analyzed dependencies. In this section—based on the data of our previous evaluation—we provide quantitative and qualitative discussions on the number and importance of the types inferred by our type inference heuristic (effectiveness).

In the evaluation presented in Chapter 4, some architectural violations are only detected due to the proposed type inference heuristic. Therefore, in this section, we investigate in the three previously evaluated systems the number of types and violations that are only inferred and detected, respectively, due to this heuristic.

5.1.1 Research Questions

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

RQ #1 – How many types are only inferred due to the proposed type inference heuristic?

RQ #2 – How many violations are only detected due to the proposed type inference heuristic?

5.1.2 Dataset

The study presented in this chapter also relies on the three systems—namely `Dito Social`, `Tim Beta`, and `PLC Attorneys`—considered in Chapter 4. We choose to use these systems because it is not a trivial task to obtain access to real-world systems.

5.1.3 Results and Discussion

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

RQ #1: How many types are only inferred due to our heuristic?
`ArchRuby` relies on static code analysis to extract the dependencies that should be verified according to the planned architecture. The type inference heuristic used by `ArchRuby` 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 three systems, enabling and disabling the type propagation heuristic. In the presented evaluation, it is important to distinguish: (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 3.6. 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 code in Listing 5.1. There are seven language features—class definition (line 1), method definition (line 2), variable assignment (line 3), object instantiation (line 3), conditional test (line 4), method invocation (line 4), and variable assignment (line 5). There are two dependencies to be inspected by the conformance process, a instantiation of class `Z` (line 3) and a method call from class `Test` to type `Z` (line 4). Finally, there is only one inferred type ([`Test`::`bar`, `x`, `Z`]).

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

Listing 5.1: Example to describe language features, dependencies, and inferred types

Results and Discussion: Table 5.1 reports our results. On average, the percentage of additional types is 4.59% (i.e., types inferred only by the proposed heuristic). `PLC Attorneys` is the only system that presented percentage below 5%, probably because it is in the early stages of its development.

Table 5.1: Number of types inferred by the proposed type propagation heuristic

Project	**# of inferred**	**# of inferred**	**% added**
	types	**types**	
	without heuristic	**with heuristic**	
Dito Social	566	598	5.65% (+32)
Tim Beta	672	709	5.51% (+37)
PLC Attorneys	154	158	2.60% (+4)
Average			**4.59%**

To provide an answer in a large context, we replicated the study in a dataset with 28 open-source Ruby systems, as described in Appendix A. As reported in Table 5.2, the percentage of additional types is 5.03% ± 3.95% (average ± standard deviation). Statistically, the number of additional types should fall between 3.50% and 6.56% within a 95% confidence interval.

After an analysis of these results, we observed that the proposed heuristic can infer more types if it also propagates the return type of method invocations. For instance, assume the piece of code in Listing 5.2. In this example, set `TYPES` would contain tuples [`Clazz`::`foo`, `y`, `A`] (line 4) and [`Clazz`::`bar`, `z`, `B`] (line 9). However, if it

Table 5.2: Number of inferred types by the proposed type inference 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
Average	**936.68**	**970.89**	**5.03**
Std Dev	**1,709.14**	**1,752.40**	**3.95**

could also infer [`Clazz::foo, x, B`] (line 3) by analyzing the type returned by method `bar`, it would 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.

```
class Clazz
  def foo
    x = bar()
    y = A.new
    y.qux(x)
  end

  def bar
    z = B.new
    z.baz
    z
  end
end
```

Listing 5.2: Example of a potential improvement in the type inference heuristic

Summary: The proposed heuristic increases the number of inspected dependencies by 5% on average, but it can increase up to 17% (for the Octopress system, in Table 5.2). We also argue that the number of additional types detected by the heuristic depends on the underlying programming style. For example, the heuristic achieves better results when developers largely rely on the dependency injection design patterns [Metz, 2012].

RQ #2: How many violations are only detected due to the proposed heuristic?

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

Methodology: In order to measure the effectiviness of the type inference heuristic, i.e., the number of violations that are identified exclusively by this heuristic, we re-performed `ArchRuby` on the three systems previously evaluated in Section 4.1, enabling and disabling the type propagation heuristic.

Results and Discussion: Three out of 48 architectural violations detected in the three systems, are detected due exclusively to the type inference heuristic. Therefore, a first analysis may point out the ineffectiveness of this heuristic. However, we claim that the heuristic has its value. For example, we found 24 violations in `Dito Social`. On one hand, from the 566 inferred types without the 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 in `Tim Beta`. From the 672 inferred types without the heuristic, we found 21 violations (3.13%); on the other hand, from the 37 inferred types by the heuristic, we could find one more violation (2.70%).

Listing 5.3 illustrates one of the violations that could be detected exclusively by the proposed type inference heuristic. The code is from `Tim Beta` and belongs to module `models`. 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 violation in line 16, `ArchRuby` could detect it exclusively 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).

```ruby
 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                                   :secundary_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
```

Listing 5.3: Divergence detected in `Tim Beta` due to the type propagation heuristic

Summary: 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 the proposed type inference heuristic is 4.35% (3/69), while in the remainder of the system it is 3.47% (43/1,238).

5.2 Comparison with Dynamic Techniques

As described in Section 2.5, type inference can be performed by dynamic analysis, which requires the execution of the system and may have a high computational cost. By contrast, dynamic techniques have an advantage to access information that is generated by the language virtual machine itself. In that sense, this section evaluates the accuracy of types inferred by static analysis when compared with types inferred with dynamic analysis. Our goal is to analyze the types inferred by dynamic analysis to suggest improvements to our proposed type inference algorithm.

5.2.1 Research Questions

This study aims to answer the following research questions:

RQ #1 – What is the accuracy of static type inference approaches compared with dynamic ones?
RQ #2 – How to improve the accuracy of static approaches?

5.2.2 Methodology

Dataset: To answer the proposed research questions, the type inference algorithm was evaluated in six open-source Ruby systems. The selection criteria were: (i) systems that have automated tests and (ii) systems compatible with version 2.1 or higher of Ruby, once the runtime can be inspected only from those versions. Table 5.3 presents the selected systems, as well as information about the percentage of test coverage[1], a reference of the last commit, and the number of lines of code.

Oracle: The types inferred during the execution of the automated tests of the selected systems (i.e., dynamic technique) were considered as an oracle. To collect these types, we implemented a profiler that stores type information generated at run time. Although this decision can generate false negatives, we selected systems with a high test coverage to minimize the problem (average of 89%, as reported in Table 5.3).

[1]This information was computed using SimpleCov tool: https://github.com/colszowka/simplecov.

Table 5.3: Evaluated systems

System	Test Coverage	Commit reference	LOC
Capistrano	94%	8a33c00	1,779
CarrierWave	81%	6c5941f	2,636
Devise	97%	4c3838b	3,683
Resque	84%	c295da9	2,315
Sass	95%	64c5c11	13,609
Vagrant	87%	0489188	8,429
Average	**89%**		**5,408**

Metrics: Type information was collected in two ways: (i) by executing automated tests (i.e., by inspecting the runtime and storing all types detected during the execution) and (ii) by executing the type inference algorithm described in Section 3.6. We generated ordered pairs to represent the collected types. Pair $(\mathtt{x}, \{\mathtt{A}\})$ denotes that variable `x` can assume type `A`.

We rely on two metrics to compare the results. The first one, named $Recall_1$ and defined in Equation 5.1, measures the number of types the proposed static type inference algorithm matches exactly when compared with types obtained during the execution of automated tests.

$$Recall_1 = \frac{\mid Static \cap Dynamic \mid}{\mid Dynamic \mid} \tag{5.1}$$

To illustrate $\mathtt{Recall_1}$, assume the following tuples are collected during automated test execution: $(\mathtt{a}, \{\mathtt{X}\})$, $(\mathtt{b}, \{\mathtt{Y}\})$, and $(\mathtt{c}, \{\mathtt{W}, \mathtt{Z}\})$. Next, assume that tuples $(\mathtt{a}, \{\mathtt{X}\})$, $(\mathtt{b}, \{\mathtt{Y}\})$ and $(\mathtt{c}, \{\mathtt{W}\})$ are obtained by the static algorithm in evaluation. Thus, $\mathtt{Recall_1}$ would be 67% (2/3), once only tuples $(\mathtt{a}, \{\mathtt{X}\})$ and $(\mathtt{b}, \{\mathtt{Y}\})$ are exactly the same in both approaches. Additionally, we propose a second metric, named $Recall_2$ and defined in Equation 5.2. This metric measures the number of variables the static algorithm is able to infer a type to, even if partially. In Equation 5.2, function *variables* returns the set of variables that has at least one inferred type.

$$Recall_2 = \frac{\mid variables(Static) \cap variables(Dynamic) \mid}{\mid Dynamic \mid} \tag{5.2}$$

Reconsider the previous example used to illustrate the calculation of $\mathtt{Recall_1}$. It is important to note that the proposed static type inference algorithm inferred only type $\{\mathtt{W}\}$ to variable `c`. Thus, the value of $\mathtt{Recall_2}$ is 100% (3/3) since the evaluated algorithm inferred at least one type to all three variables (`a`, `b`, and `c`).

5.2.3 Results and Discussion

RQ #1 – What is the accuracy of static type inference approaches compared with dynamic ones?

After the execution of the automated tests and the static algorithm under evaluation, we calculate the recall measures described in the previous section. Table 5.4 reports the results to the six evaluated systems. On average, the value for $\texttt{Recall}_1$ is 44.4%; on the other hand, for $\texttt{Recall}_2$ the value rises to 58.1%. The values in parentheses represent the numerator and denominator of the $\texttt{Recall}_1$ and $\texttt{Recall}_2$ equations. The `CarrierWave` system obtained 50.8% for $\texttt{Recall}_1$, which was the highest measure. This value corresponds to 62 out of 122 variables that have their types inferred by the static approach. The `Sass` system obtained 58.4% for $\texttt{Recall}_2$, 635 out of 1,088 variables have at least one type inferred by the static approach.

Table 5.4: Results comparing static and dynamic techniques

System	$\boldsymbol{Recall_1}$	$\boldsymbol{Recall_2}$
Capistrano	39.0% (30/77)	45.5% (35/77)
CarrierWave	50.8% (62/122)	57.4% (70/122)
Devise	43.8% (155/354)	56.5% (200/354)
Resque	45.0% (9/20)	55.0% (11/20)
Sass	46.0% (500/1,088)	58.4% (635/1,088)
Vagrant	40.8% (220/539)	55.7% (300/539)
Average	**44.4%**	**58.1%**

Listing 5.4 illustrates a scenario where the proposed static type inference algorithm was not capable to infer the type of a variable for the `Vagrant` system. The loop in lines 2-4 iterates over an array to assign a value to variable `command`. This array is populated by method `create_command_filters`, which uses reflection to produce the array elements (lines 11-13), and hence the values will be known only during run time. Therefore, it is very complex to static analysis to infer the type of these elements.

```ruby
def filter(command)
  command_filters.each do |c|
    command = c.filter(command) if c.accept?(command)
  end
  command
end
def create_command_filters
  [].tap do |filters|
    @@cmd_filters.each do |cmd|
      require_relative "command_filters/#{cmd}"
      class_name = "VagrantPlugins::CommunicatorWinRM::
             CommandFilters::#{cmd.capitalize}"
      filters << Module.const_get(class_name).new
    end
  end
end
```

Listing 5.4: Example code of the Vagrant system

Similarly, Listing 5.5 illustrates a second example where it was not possible to detect the type of a variable of the Capistrano system. Variable `value` is initialized in line 2. However, its value is also updated in a *loop* (lines 3-5). Similarly to the previous example, `value` has its value defined dynamically (in the *loop*), which imposes complexity to type inference through static analysis.

```ruby
1 def fetch(key, default=nil, &block)
2   value = fetch_for(key, default, &block)
3   while callable_without_parameters?(value)
4     value = set(key, value.call)
5   end
6   return value
7 end
```

Listing 5.5: Example code of Capistrano

Finally, Listing 5.6 shows a scenario at the Devise system where the value of a variable is obtained through the access of a `Hash` data structure. In line 5, the `skip` variable receives the value stored in key `:skip_helpers` of the `options` hash. Therefore, a static analysis should infer the values passed as key and value to this structure to correctly determine the type assigned to `skip`, which is not a trivial task.

```ruby
def default_used_helpers(options)
  singularizer = lambda { |s| s.to_s.singularize.to_sym }
  if options[:skip_helpers] == true
    @used_helpers = @used_routes
  elsif skip = options[:skip_helpers]
    @used_helpers = self.routes - Array(skip).map(&singularizer)
  else
    @used_helpers = self.routes
  end
end
```

Listing 5.6: Example code of Devise system

RQ #2 – How to improve the accuracy of static approaches?

To answer this second question, we manually analyzed the variables whose types could not be inferred by the proposed static type inference algorithm. More specifically, we investigated each code related to each variable to understand the reasons that led static analysis to fail. This analysis provided us with suggestions of improvements that could be applied to the algorithm to increase its accuracy. The proposed improvements are classified with respect to their complexity:

- *Simple*: Type inference heuristics should consider the `include` and `extend` features supported by Ruby, which offer the support of *mixins* to implement multiple inheritance [Bracha and Cook, 1990]. For instance, suppose that a class `A` includes module `B`, i.e., all methods defined in `B` are also part of class `A`. Therefore, static analysis algorithms should simulate the functioning of those features by inspecting all methods defined in modules that were added in classes through the use of `include` and `extend`.

- *Moderate*: Type inference heuristics should consider the execution and return of blocks, which are first class functions in Ruby and can be stored in variables and used as parameters to function calls. For example, a block could change or define a variable type during or after returning its execution. Basically, a block is like a method, although there are differences in the context of execution. In short, static type inference algorithms should track the changes that are made by the execution of blocks.

- *Complex*: Type inference heuristics should consider values stored in data structures, such as `Array`, `Hash`, `Set`, etc. Ruby supports reflection and dynamic evaluation of code (e.g., eval), therefore the type inference algorithm should be able to analyze dynamic instructions. For example, `instance_eval`, `class_eval`, and `define_method` are features that change the program at run time. Also, the algorithm should consider values assigned to instance variables and hence static analysis needs to be aware of the execution context and the assignment flow.

Table 5.5 reports the results of a manual analysis of the impact of the proposed changes in the evaluated systems. The adoption of the simple and moderate improvements can raise the average $\texttt{Recall}_1$ to 78.9% (44.4+2.0+0.7+13.4+18.4), which is a considerable increase to the static analysis. Moreover, by adding the complex modifications, this value would increase to 100% (78.9+11.7+3.5+5.9). As another significant result, if only moderate improvements (block return and block execution) are implemented, $\texttt{Recall}_1$ would raise from 44.4% to 76.2%.

Table 5.5: Impact of proposed improvements in the $\texttt{Recall}_1$ results

	***Recall*₁**								
	Trivial	**Simple**		**Moderate**		**Complex**			
Systems	**base**	**include**	**extend**	**block return**	**block execution**	**values in structures**	**dynamic instruc.**	**execution flow in instance variables**	**Total**
Capistrano	39.0%	5.2%	1.3%	14.3%	13.0%	15.6%	1.3%	10.3%	**100%**
CarrierWave	50.8%	6.6%	1.6%	11.5%	13.1%	9.8%	2.5%	4.1%	**100%**
Devise	43.8%	2.8%	1.4%	12.7%	15.5%	9.0%	6.3%	8.5%	**100%**
Resque	45.0%	0%	0%	20.0%	15.0%	10.0%	0%	10.0%	**100%**
Sass	46.0%	0.6%	0.6%	13.8%	18.4%	12.9%	3.6%	4.1%	**100%**
Vagrant	40.8%	2.8%	0.4%	13.0%	22.3%	11.1%	2.2%	7.4%	**100%**
Average	**44.4%**	**2.0%**	**0.7%**	**13.4%**	**18.4%**	**11.7%**	**3.5%**	**5.9%**	**100%**

5.3 Measuring the Impact of Dynamic Analysis

As outlined in the previous section, dynamic analysis techniques can infer types that are complex to infer with static analysis. Therefore, we can use type information generated by dynamic analysis to improve `ArchRuby` results. This would raise the number of dependencies to be analyzed by architectural conformance process, and possibly lead to detection of new architectural violations. Therefore, in this section, we investigate the impact of adding type information generated by dynamic analysis to the `ArchRuby` tool.

5.3.1 Research Questions

This study aims to answer the following research questions:

RQ #1 – How could `ArchRuby` be complemented with information collected with dynamic analysis?
RQ #2 – Does type information collected through dynamic analysis improve the effectiveness of architectural conformance processes?

5.3.2 Results and Discussion

RQ #1 – How could `ArchRuby` be complemented with information collected with dynamic analysis?
`ArchRuby` uses a static type inference algorithm to infer types. However, due to features present in dynamically typed languages, not all types can be statically inferred. As mentioned before, type information is very important to the architectural conformance process because it reveals dependencies between classes.

Methodology: In order to collect type information during system execution, we developed a profiler to Ruby systems. More specifically, this profiler uses an API provided by the Ruby virtual machine to inspect the run-time environment during system execution. It records all types generated during the execution. It is important to notice that the Ruby virtual machine allows run-time inspections only in versions greater than 2.1 of the language. Therefore, we selected the PLC Attorneys systems because it uses version 2.2 of Ruby.

We rely on automated tests to collect run-time information. All the information collected by the profiler is stored in a CSV file. Each line of this file contains the following columns: class, method, local variables, and the types the local variables have assumed. This file will be used later to complement type information of `ArchRuby`. To illustrate the data in this CSV file, suppose a class `Professor` with a method named `students` that has a local variable `disciplines`, which is an instance of class `Discipline`. In that sense, the content of the first column is `Professor`, the second is `students`, the third is `disciplines`, and the last one is `Discipline`. To use this information in `ArchRuby` we added a new option that can be passed to the executable, called `more_types`. The following example illustrates the usage:

```
archruby --arch_def_file=/arch_def.yml --app_root_path=/sys_path
         --more_types=/information.csv
```

Results and Discussion: We collected type information in two scenarios: executing the static algorithm and executing the automated tests. The static algorithm inferred 162 variable types and the profiler collected 238 variable types. Listing 5.7 shows an example where the static analysis technique could not infer the variable type.

```
 1 def create
 2   ...
 3   presenters = Presenters::Presenter.get_for_all_representations
 4   json_presenter = presenters[:json].new(@project)
 5   html_presenter = presenters[:html].new(@project)
 6 
 7   format.html { redirect_to html_presenter.redirect_path }
 8   format.json { render json: json_presenter.success.render }
 9   ...
10 end
```

Listing 5.7: Example of variable type discovered by dynamic analysis

Local variables `json_presenter` and `html_presenter` (lines 4-5) have their types defined by accessing the value stored in the `presenters` hash (line 3). As described in Section 5.2, it is not trivial to static analysis to infer the types of such variables. More specifically, it is considered complex once the static algorithm has to map values stored in structures. Therefore, we can use type information generated by dynamic analysis to complement `ArchRuby` execution.

RQ #2 – Does type information collected through dynamic analysis improve the effectiveness of architectural conformance processes?

The previous research question indicated that dynamic analysis could be used to complement type information generated by static analysis. Nonetheless, it is important to investigate if these additional types indeed contribute to the architectural conformance process. In other words, if these new types lead to the detection of new architectural violations.

Methodology: We use the CSV file generated by the profiler described in the previous research question. We also use the PLC system (these violations have already been described in Section 4.5). To answer the second research question, we execute `ArchRuby` with and without the additional type information. In each step, we collected the total number of detected violations.

Results and Discussion: When configured to run only with types that are inferred statically, `ArchRuby` identified two architectural violations in the PLC system. On the other hand, when type information provided by the profiler was also considered `ArchRuby` detected four architectural violations.

Listing 5.8 presents the code responsible for the two architectural violations detected only due to the types inferred by dynamic profiler.

```
1 def can_access?(admin)
2   mailers = ["AdminNotifier", "MasterNotifier"]
3   notifier_instance = nil
4   mailers.each do |mailer|
5     notifier_instance = Object.const_get(mailer).new
6     notifier_instance.send_notification(admin.id)
7   end
8   admin_projects.where(admin_id: admin.id).present?
9 end
```

Listing 5.8: Code of PLC system

The array assigned to `mailers` (line 2) contains the name of two classes responsible to notify admins and master users when another user tries to access a resource. The classes names are provided as strings. In line 5, the string is used to create a new object. The instantiation of a new object is possible due to the reflexion services supported by Ruby. However, this represents a limit to static code analysis since the type of `notifier_instance` is only know at run time. Since the type information generated by dynamic analysis complements the number of dependencies analyzed by `ArchRuby`, the tool can detect more architectural violations.

5.4 Final Remarks

In this chapter, we conducted an evaluation on the type inference heuristic that provided evidences of the increase on the number of generated dependencies to be analyzed by the architectural conformance process. Besides that, we conducted a study to compare static techniques, which the proposed type inference algorithm is based on, with the dynamic techniques. Complementary, we measure the impact of adding type information generated by dynamic analysis to `ArchRuby`.

In conclusion, our heuristic increased the number of inspected dependencies by 5% on average. The use of such heuristic led to 4.35% of the violations being identified exclusively by our heuristic. We also showed that, on average, the proposed type inference algorithm could infer 44.4% of the types in Ruby systems when compared to dynamic techniques. Moreover, we pointed out seven improvements that can be incorporated in the initially proposed type inference heuristic to raise its accuracy. Finally, we presented a study that demonstrates `ArchRuby` can benefit from type information generated by dynamic analysis. For this study, we included the possibility to execute `ArchRuby` with type information generated by dynamic analysis. Furthermore, `ArchRuby` found two more violations for the `PLC Attorneys` system due to type information generated by dynamic analysis.

Chapter 6

Conclusion

The architectural design can directly affect software performance, robustness, portability, and maintainability since it comprises a set of key decisions and best practices that enable the software evolution [Passos et al., 2010; Murphy et al., 1995]. However, as a project evolves, these architectural patterns tend to deteriorate—due to lack of knowledge, short deadlines, etc.—and hence nullifying the benefits provided by architectural designs. To tackle this problem, several techniques have been proposed, such as Reflexion Models [Murphy et al., 1995], Dependency Structure Matrices [Sullivan et al., 2001], Dependency Constraint Languages [Terra and Valente, 2009], ArchLint [Maffort et al., 2013], etc. Nevertheless, none of them addresses the particularities of dynamically typed languages.

To tackle this problem, we describe in this master dissertation an approach to perform architectural conformance and to better visualize the architecture of systems implemented in dynamically typed languages. Particularly, our approach, called `ArchRuby`, targets systems implemented in the Ruby language. In a nutshell, `ArchRuby` receives as input the architectural rules (defined by means of a DSL) and the source code of the target system. Thereafter, it triggers the architectural conformance process in order to detect design decisions that do not respect the intended architecture. Moreover, `ArchRuby` implements a type inference heuristic to raise the number of dependencies to be analyzed in the conformance process.

To evaluate the proposed approach, we conducted four evaluations. First, we conducted an evaluation with the architects of three real-word systems and `ArchRuby` found 48 architectural violations the developers had no prior knowledge. It also provided developers with two high-level views of the software architecture that help them to reason about the system organization. Second, we measure the effectiveness of the proposed type inference heuristic reporting that (i) the number of analyzed types raises 5% on average and (ii) certain violations are only detected due to this heuristic. Third, we conducted a study to compare the proposed type inference algorithm with dynamic techniques showing that (i) the proposed type inference algorithm provides an average recall of 44% and (ii) seven improvements can be implemented in static techniques to raise the number of inferred types. Fourth and last, we conducted a study with a real-world system adapting `ArchRuby` to use information generated by dynamic analysis techniques in order to raise the number of analyzed dependencies.

We organized this chapter as follows. First, Section 6.1 reviews the contributions of our research. Next, Section 6.2 points the limitations of our approach. Finally, Section 6.3 describes further work.

6.1 Contributions

This research makes the following contributions:

- The design of an architectural conformance approach that targets dynamically typed languages. This approach, called `ArchRuby`, includes the definition of a DSL to specify the architectural rules of a system, two high-level models to better visualize the system architecture, and a type inference heuristic to increase the number of evaluated dependencies (Chapter 3).
- A prototype tool that supports our approach and hence performs the architectural conformance process for systems implemented in Ruby. It receives as input the architectural rules and the source code of the target system to output the architectural violations and two high-level architectural models (a graph and a DSM) to reason about the system organization and to better visualize the identified violations.
- An evaluation of the proposed approach with software architects of three real-world systems—`Dito Social`, `Tim Beta`, and `PLC Attorneys`. As the result, `ArchRuby` found 48 architectural violations that developers had no prior knowledge. The architects also reported that the violations negatively impact on the maintenance of the systems (Chapter 4).
- A study evaluating the proposed type inference algorithm in three real-world systems reporting that it increases the number of dependencies to be analyzed and helps to find architectural violations that cannot be detect without it. Moreover, we identify that our algorithm needs to be improved and, in that sense, we adapt `ArchRuby` to be complemented with type information collected with dynamic analysis.

6.2 Limitations

Our work has the following limitations:

- Our DSL to specify architectural rules does not allow architects to formulate fine grained rules. For example, it does not support a rule that precludes the instantiation of a class by a specific module. To tackle such case, the proposed DSL needs to be improved to address the features offered by object-oriented languages, such as object instantiations.
- The proposed heuristic to infer types does not cover all possible scenarios, such as instance variable assignment flow. We also reported the limitations of our heuristic in Section 5.2.
- We have not evaluated whether our approach provides equivalent results in system implemented for contexts different from web-based systems.

6.3 Future Work

We consider that our work can be complemented with the following future work:

- Proposed Approach: (i) incorporating an architectural repair solution that provides suggestions on how to solve the detected violations, as implemented by ArchFix [Terra et al., 2015] to Java systems; (ii) improving our type propagation heuristic by implementing the improvements listed in Section 5.2; and (iii) integrating the proposed approach to mainstreams IDEs (e.g., RubyMine) for a better usability and to be continuously used by developers during development phase.
- The `ArchRuby` Tool: the tool can be extended at least as follows: (i) by executing automated tests when they exist. `ArchRuby` could automatically detect and execute the tests of the system under analysis to complement type information that is extracted by static analysis; and (ii) by integrating with systems that automated continuous integration tasks; and (iii) by extending the tool to other dynamically typed languages. For example, we can implement modules to work specifically with other languages and `ArchRuby` would automatically detect which module to use based on the system under analysis.
- Evaluation with new systems: `ArchRuby` can be evaluated in more systems implemented in Ruby, specially in systems implemented for contexts different from web-based systems.

Bibliography

Agesen, O. and Holzle, U. (1995a). Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *10th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 91–107.

Agesen, O. and Holzle, U. (1995b). Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. Technical report.

Agesen, O., Palsberg, J., and Schwartzbach, M. I. (1995). Type inference of self: Analysis of objects with dynamic and multiple inheritance. *Software: Practice and Experience*, 25(9):975–995.

Ascher, D. and Lutz, M. (1999). *Learning Python.* O Reilly Media.

Baldwin, C. Y. and Clark, K. B. (1999). *Design Rules: The Power of Modularity.* MIT Press.

Bean, M. (2015). *Laravel 5 Essentials.* Packt Publishing.

Black, D. A. (2009). *The Well-Grounded Rubyist.* Manning.

Borchers, J. (2011). 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.

Bosch, J. (2004). Software architecture: The next step. In *First European Workshop (EWSA)*, pages 194–199.

Bracha, G. and Cook, W. (1990). Mixin-based inheritance. In *5th Conference on Object-Oriented Programming: System Languages, and Applications (OOPSLA)*, pages 303–311.

Brunet, J., Guerreiro, D., and Figueiredo, J. (2011). Structural conformance checking with design tests: An evaluation of usability and scalability. In *27th International Conference on Software Maintenance (ICSM)*, pages 143–152.

Converse, T. (2002). *PHP Bible*. Wiley.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co.

De Schutter, K. (2012). Automated architectural reviews with semmle. In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 557–565.

Flanagan, D. (2006). *JavaScript: The Definitive Guide*. O Reilly Media.

Furr, M., hoon (David) An, J., Foster, J. S., and Hicks, M. (2009). Static type inference for Ruby. In *24th Symposium on Applied Computing (SAC)*, pages 1859–1866.

Gabriel, R. P. (1986). *Performance and Evaluation of LISP Systems*. MIT Press.

Goldberg, A. and Robson, D. (1989). *Smalltalk-80: The Language and its Implementation*. Addison-Wesley.

Greenfeld, D. R. and Greenfeld, A. R. (2015). *Two Scoops of Django: Best Practices for Django 1.8*. Two Scoops Press.

Hartl, M. (2012). *Ruby on Rails Tutorial: Learn Web Development with Rails*. Addison-Wesley.

Johnson, R. E. (1986). Type-checking smalltalk. In *11th Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA)*, pages 315–321.

Jones, R. and Lins, R. D. (1996). *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley.

Knodel, J., Muthig, D., Haury, U., and Meier, G. (2008a). Architecture compliance checking - experiences from successful technology transfer to industry. In *12th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 43–52.

Knodel, J., Muthig, D., Naab, M., and Lindvall, M. (2006). Static evaluation of software architectures. In *10th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 279–294.

Knodel, J., Muthig, D., and Rost, D. (2008b). Constructive architecture compliance checking - an experiment on support by live feedback. In *24th International Conference on Software Maintenance (ICSM)*, pages 287–296.

Maffort, C., Valente, M. T., Anquetil, N., Hora, A., and Bigonha, M. (2013). Heuristics for discovering architectural violations. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 222–231.

Maffort, C., Valente, M. T., Terra, R., Bigonha, M., Anquetil, N., and Hora, A. (2016). Mining architectural violations from version history. *Empirical Software Engineering Journal*, pages 1--42.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3:184–195.

Metz, S. (2012). *Practical Object-Oriented Design in Ruby: An Agile Primer*. Addison-Wesley.

Miranda, S., Rodrigues, E., Valente, M. T., and Terra, R. (2016a). Architecture conformance checking in dynamically typed languages. *Journal of Object Technology*, 15(3):1–34.

Miranda, S., Valente, M. T., and Terra, R. (2015a). ArchRuby: Conformidade e visualização arquitetural em linguagens dinâmicas. In *VI Brazilian Conference on Software: Theory and Practice (CBSoft), Tools Session*, pages 17–24.

Miranda, S., Valente, M. T., and Terra, R. (2015b). Conformidade e visualização arquitetural em linguagens dinâmicas. In *XVIII Ibero-American Conference on Software Engineering (CIbSE), Software Engineering Technologies (SET) Track*, pages 137–150.

Miranda, S., Valente, M. T., and Terra, R. (2016b). Inferência de tipos em ruby: Uma comparação entre técnicas de análise estática e dinâmica. In *IV Workshop on Software Visualization, Evolution and Maintenance (VEM)*, pages 105–112.

Murphy, G., Notkin, D., and Sullivan, K. (1995). Software reflexion models: Bridging the gap between source and high-level models. In *3rd Symposium on Foundations of Software Engineering (FSE)*, pages 18–28.

Palsberg, J. and Schwartzbach, M. I. (1991). Object-oriented type inference. In *6th Conference on Object-Oriented Programming: System Languages, and Applications (OOPSLA)*, pages 146–161.

Passos, L., Terra, R., Diniz, R., Valente, M. T., and Mendonça, N. (2010). Static architecture-conformance checking: An illustrative overview. *IEEE Software*, 27(5):82–89.

Perrotta, P. (2010). *Metaprogramming Ruby: Program Like the Ruby Pros*. Pragmatic Bookshelf.

Resig, J. and Bibeault, B. (2013). *Secrets of the JavaScript Ninja*. Manning Publications.

Sangal, N., Jordan, E., Sinha, V., and Jackson, D. (2005). Using dependency models to manage complex software architecture. In *20th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 167–176.

Sarkar, S., Ramachandran, S., Kumar, G. S., Iyengar, M. K., Rangarajan, K., and Sivagnanam, S. (2009). Modularization of a large-scale business application: A case study. *IEEE Software*, 26:28–35.

Schwartz, R. L. (2011). *Learning Perl*. O Reilly Media.

Sommerville, I. (2010). *Software Engineering (9th Edition)*. Pearson.

Sullivan, K. J., Griswold, W. G., Cai, Y., and Hallen, B. (2001). The structure and value of modularity in software design. In *9th International Symposium on Foundations of Software Engineering (FSE)*, pages 99–108.

Sussman, G. J. and Guy L. Steele, J. (1998). Scheme: A interpreter for extended lambda calculus. *Higher Order Symbol. Comput.*, 11:405–439.

Terra, R. and Valente, M. T. (2009). A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 32(12):1073–1094.

Terra, R., Valente, M. T., Czarnecki, K., and Bigonha, R. S. (2015). A recommendation system for repairing violations detected by static architecture conformance checking. *Software: Practice and Experience*, 45(3):315–342.

Thomas, D., Fowler, C., and Hunt, A. (2004). *Programming Ruby The Pragmatic Programmers' Guide*. Pragmatic Bookshelf.

TIOBE index (2017). `http://www.tiobe.com/tiobe-index`.

Virding, R., Wikstrom, C., Williams, M., and Armstrong, J. (1996). *Concurrent Programming in Erlang*. Prentice Hall.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer.

Appendix A

Open-source Dataset

Table A.1 summarizes information about our open-source dataset. It contains 28 out of the 30 most starred Ruby projects in GitHub (on August, 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.[1] 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[2] file with a list of technology web sites. In total, we analyzed over half million LOC and eight thousand `rb` files.

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

[2] Outline Processor Markup Language (OPLM) is an XML format for outlines, which is straightforward imported by RSS readers.

Table A.1: 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 (vlatestes-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

Appendix B

Dito Social

B.1 Architectural Rules

```
helpers:
  files: 'app/helpers/**/*.rb, app/helpers/events/**/*.rb'

badge_controller:
  files: 'app/controllers/badge/**/*.rb'
  allowed: 'resque, badge_workers, controller_base'

badge_workers:
  files: 'app/workers/badge/**/*.rb, lib/send_badge_to_social_badge.rb'
  allowed: 'resque, badge_model, user_model, social_helper'

badge_model:
  files: 'app/models/badge.rb, app/models/user_badge.rb'
  required: 'activerecord'
  allowed: 'badge_workers, resque, app_model, activerecord'

dashboard_controller:
  files: 'app/controllers/dashboard/**/*.rb'
  allowed: 'dashboard_finder, stats_model, network_model, action_model, app_model,
      interaction_model, post_model, social_helper, user_network_model, controller_base,
      referral_model, origin_model, http_party, user_agent_model, user_model, airbrake'

dashboard_finder:
  files: 'app/models/dashboard/**/*.rb'
  allowed: 'stats_model, origin_model, action_model, app_model, user_model, social_helper,
      user_agent_model, event_model, app_user_network_model, network_model,
      user_network_model'

stats_generator:
  files: 'app/models/generate_stats/**/*.rb'
  allowed: 'stats_model, event_model, action_model, network_model'

stats_model:
  files: 'app/models/action_source_stats.rb, app/models/action_stats.rb,
     app/models/revenue_source_stats.rb, app/models/revenue_user_stats.rb,
     app/models/stats.rb, app/models/total_stats.rb,
     app/models/user_agent_platform_type_stats.rb'
  allowed: ''

events_controller:
  files: 'app/controllers/events/**/*.rb'
  allowed: 'resque, app_model, social_helper, events_workers, event_model, controller_base'

events_workers:
  files: 'app/workers/events/**/*.rb, lib/post_event_job.rb'
  allowed: 'resque, social_helper, user_model, user_network_model, event_creator,
     referral_model, app_model, network_model'

event_creator:
  files: 'app/models/create_event.rb'
  allowed: 'social_helper, event_model, action_model, app_model, target_model, solr_workers,
     resque, referrer_model, origin_model, referral_model, user_network_model, user_model,
     network_model, user_agent_model'

ranking_controller:
  files: 'app/controllers/ranking/**/*.rb'
  allowed: 'resque, ranking_workers, controller_base'

ranking_workers:
  files: 'app/workers/ranking/**/*.rb'
  allowed: 'resque, app_model, ranking_model'

referral_controller:
  files: 'app/controllers/referral/**/*.rb'
  allowed: 'referral_model, controller_base, airbrake'

share_controller:
  files: 'app/controllers/share/**/*.rb'
  allowed: 'resque, post_model, post_workers, facebook_info_retriever, social_helper,
     controller_base, http_party, interaction_model, airbrake, rails'

post_workers:
  files: 'lib/customer_post_info_job.rb, lib/customer_post_first_update_job.rb,
     lib/customer_post_update_job.rb, lib/customer_send_interactions_to_redis.rb,
     lib/save_in_redis.rb, lib/update_big_post.rb, lib/prioritize_posts.rb, lib/post_info_job.rb,
     lib/post_update_job.rb, lib/send_top_agents_to_redis.rb'
  allowed: 'resque, post_model, social_helper, map_post_keys, post_summary,
     interaction_model'

login_controller:
  files: 'app/controllers/social_login/**/*.rb'
  allowed: 'app_model, user_model, user_network_model, login_workers, resque,
     controller_base, app_user_network_model, network_model, airbrake'

login_workers:
  files: 'app/workers/login/**/*.rb, lib/update_friends_job.rb, lib/save_user.rb'
  allowed: 'resque, user_model, social_helper, friendship_model'

solr_workers:
  files: 'app/workers/solr/**/*.rb'
  allowed: 'resque, social_helper, solr, app_model, app_user_network_model, json,
     action_model, rails'

report_controller:
  files: 'app/controllers/reports_controller.rb'
  allowed: 'resque, report_model, reports_workers, interaction_model, controller_base,
     report_model'

reports_workers:
  files: 'app/workers/reports/**/*.rb'
  allowed: 'resque'

notification_controller:
  files: 'app/controllers/notification/**/*.rb'
  allowed: ''

controller_base:
  files: 'app/controllers/application_controller.rb'

resque:
  gems: 'Resque'

social_helper:
  gems: 'SocialHelper'

view:
  files: 'app/views/**/*.rb'
  forbidden: 'model'

report_model:
  files: 'app/models/report/**/*.rb'
  allowed: 'post_model, social_helper, interaction_model, rails, mail, aws, http_party'

app_model:
  files: 'app/models/app.rb'
  required: 'activerecord'

event_model:
  files: 'app/models/event.rb'
  allowed: 'action_model'
  required: 'activerecord'

action_model:
  files: 'app/models/action.rb'
  required: 'activerecord'

user_network_model:
  files: 'app/models/user_network.rb'
  required: 'activerecord'

network_model:
  files: 'app/models/network.rb'

app_user_network_model:
  files: 'app/models/app_user_network.rb'
```

```

user_model:
  files: 'app/models/user.rb'
  required: 'activerecord'
  allowed: 'user_infos, network_connections_log_model, user_network_model, login_workers,
      resque, app_model, social_helper, app_user_network_model, facebook_info_retriever, json,
      facebook, network_model, event_model, action_model, interaction_model'

friendship_model:
  files: 'app/models/friendship.rb'

user_infos:
  files: 'app/models/city.rb, app/models/language.rb, app/models/education.rb'
  required: 'activerecord'

referral_model:
  files: 'app/models/referral.rb'
  required: 'activerecord'

origin_model:
  files: 'app/models/origin.rb, app/models/utm.rb, app/models/utm_medium.rb,
      app/models/utm_campaign.rb, app/models/source.rb'
  required: 'activerecord'

ranking_model:
  files: 'app/models/ranking.rb'
  required: 'activerecord'

target_model:
  files: 'app/models/target.rb, app/models/target_type.rb'
  required: 'activerecord'

post_model:
  files: 'app/models/post.rb'
  required: 'activerecord'
  allowed: 'resque, post_workers, post_logger, facebook_info_retriever, social_helper,
      interaction_model, question_option_model, logger, activerecord, rails'

user_agent_model:
  files: 'app/models/user_agent_manager.rb, app/models/user_agent.rb,
      app/models/user_agent_platform.rb'

referrer_model:
  files: 'app/models/referrer.rb'
  required: 'activerecord'

question_option_model:
  files: 'app/models/question_option.rb'
  required: 'activerecord'

map_post_keys:
  files: 'lib/map_redis_keys.rb'

post_logger:
  files: 'config/initializers/posts_logger.rb'

```

```
post_summary:
  files: 'app/models/post_summary.rb'
  required: 'activerecord'

interaction_model:
  files: 'app/models/interaction.rb, app/models/interaction_agent.rb'

facebook_info_retriever:
  files: 'lib/facebook_info_retriever.rb'
  allowed: 'facebook, airbrake'

network_connections_log_model:
  files: 'app/models/network_connections_log.rb'
  required: 'activerecord'

twitter:
  gems: 'Twitter'

json:
  gems: 'JSON'

airbrake:
  gems: 'Airbrake, Notification::Airbrake'

http_party:
  gems: 'HTTParty'

facebook:
  gems: 'Koala'

activerecord:
  gems: 'ActiveRecord'

actioncontroller:
  gems: 'ActionController'

aws:
  gems: 'AWS'

solr:
  gems: 'Sunspot'

logger:
  gems: 'Logger'

rails:
  gems: 'Rails'

mail:
  gems: 'Mail'
```

Listing B.1: `Dito Social` architectural specification file

B.2 Reflexion Model

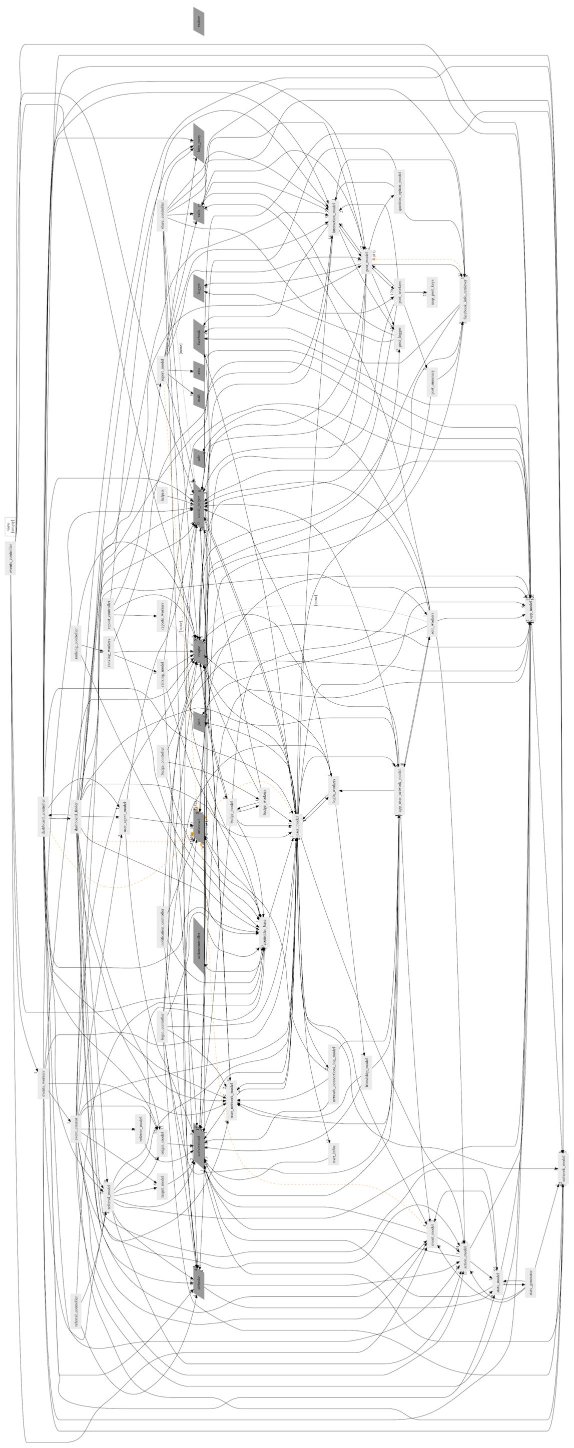


Figure B.1: RM automatically computed by `ArchRuby` for `Dito Social`

Appendix C

Tim Beta

C.1 Architectural Rules

```
controllers:
  files: 'app/controllers/**/*.rb'
  allowed: 'models, helpers, workers, core, facebook, twitter, foursquare, youtube, twitter_oauth,
      google_plus, dito_social_p, cep_finder, linker, can_can, facebook_verifier, orkut_deleter,
      resque, logistica, oauth, tim_points, logged_home, datapoints, instagram, rails, devise,
      base_controller, action_controller, spreadsheet, opensocial'

core:
  files: 'app/core/**/*.rb'
  allowed: 'models, helpers, facebook, twitter, foursquare, gmail, mailers, instagram,
      dito_social_p, twitter_oauth, contact_us, resque, sanitize, active_record, workers, hoptoad'

datapoints:
  files: 'app/core/datapoints/**/*.rb, lib/collect_datapoints.rb'

invitations:
  files: 'app/core/invitations/**/*.rb'

contact_us:
  files: 'app/core/contact_us/**/*.rb'

helpers:
  files: 'app/helpers/**/*.rb, app/helpers/admin/**/*.rb, app/helpers/api/**/*.rb,
      app/helpers/timbeta/**/*.rb, app/helpers/timbetaapp/**/*.rb,
      lib/home_controller_helper.rb, lib/network_controller_helpers.rb,
      lib/connect_link_session_attributes.rb'
  allowed: 'models, dito_social_p, tim_points, rails, linker'

mailers:
  files: 'app/mailers/**/*.rb'
  allowed: 'models, postage_app, active_record, action_mailer'

action_mailer:
  gems: 'ActionMailer'

```

```
models:
  files: 'app/models/**/*.rb'
  allowed: 'core, helpers, resque, logistica, dito_social_p, postage_app, workers, facebook, devise,
      csv, olap, twitter_oauth, datapoints, can_can, tim_points, linker, twitter, rails, active_record,
      image_magick, action_controller'

logistica:
  files: 'lib/logistics.rb'

tim_points:
  files: 'lib/tim_points.rb'

cep_finder:
  files: 'lib/cep.rb'

logged_home:
  files: 'lib/logged_home.rb'

linker:
  files: 'lib/link.rb, lib/normalize_link.rb'

views:
  files: 'app/views/**/*.rb'
  allowed: 'controllers'

workers:
  files: 'app/workers/**/*.rb'
  allowed: 'models, core, facebook, dito_social_p, rails'

facebook:
  gems: 'Koala'

orkut_deleter:
  files: 'lib/correct_orkut/**/*.rb'

facebook_verifier:
  files: 'lib/verify_token.rb, lib/facebook_token.rb'

youtube:
  gems: 'YouTubeIt'

olap:
  files: 'lib/send_user_to_olap_job.rb, lib/send_chip_to_olap_job.rb'

google_plus:
  gems: 'GooglePlus'

foursquare:
  gems: 'Foursquare2'

twitter:
  gems: 'Twitter'

instagram:
  gems: 'Instagram'

active_record:
  gems: 'ActiveRecord'

twitter_oauth:
  gems: 'TwitterOAuth'

oauth:
  gems: 'OAuth2, OAuth'

opensocial:
  gems: 'OpenSocial'

dito_social_p:
  gems: 'UserNetwork, Network, Friendship, BadgesEarner, Badge, Ranking'

base_controller:
  gems: 'ApplicationController, Api::ApplicationController'

action_controller:
  gems: 'ActionController'

resque:
  gems: "Resque"

postage_app:
  gems: "PostageApp"

csv:
  gems: "FasterCSV"

rails:
  gems: 'Rails'

devise:
  gems: 'Devise'

can_can:
  gems: 'CanCan'

gmail:
  gems: 'Gmail'

spreadsheet:
  gems: 'Spreadsheet'

image_magick:
  gems: 'Magick'

sanitize:
  gems: 'Sanitize'

hoptoad:
  gems: 'HoptoadNotifier'
```

Listing C.1: `Tim Beta` architectural specification file

C.2 Reflexion Model

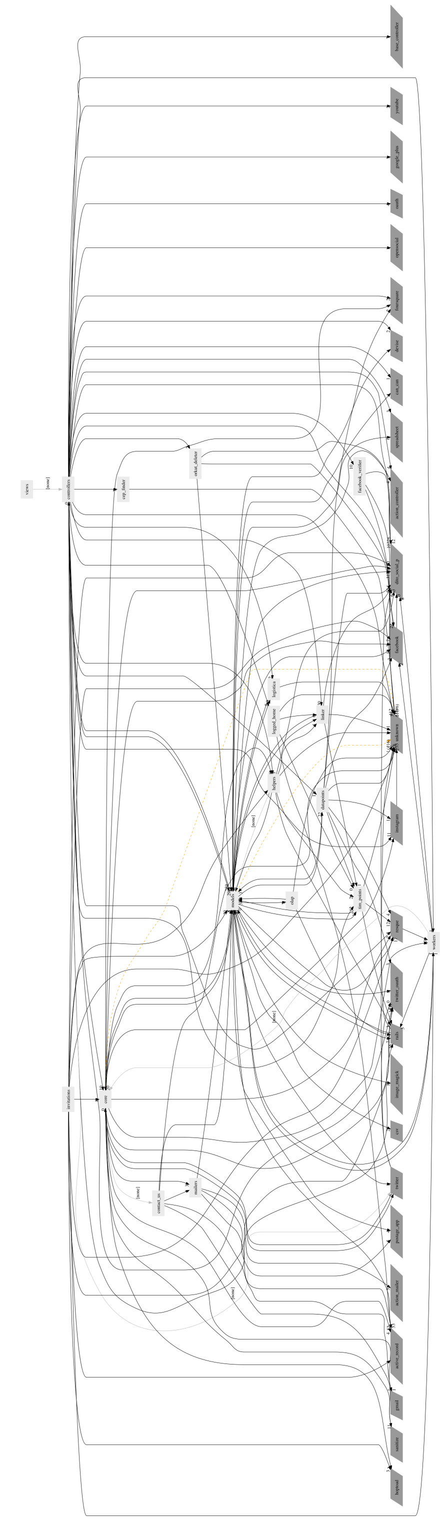


Figure C.1: RM automatically computed by `ArchRuby` for `Tim Beta`

Appendix D

PLC Attorneys

D.1 Architectural Rules

```
controllers:
  files: 'app/controllers/**/*.rb'
  allowed: 'presenters, devise, actioncontroller, project_relations, project, admins,
      consolidated_control'

project:
  files: 'app/models/project.rb'
  required: 'activerecord'
  allowed: 'chartdraw, project_relations, admins'

project_relations:
  files: 'app/models/area.rb, app/models/company.rb, app/models/areas_project.rb,
      app/models/attack.rb, app/models/control.rb, app/models/diagnostic.rb,
      app/models/improvement.rb, app/models/action.rb, app/models/task.rb,
      app/models/responsible.rb'
  required: 'activerecord'
  allowed: 'chartdraw, mailers, admins'

mailers:
  files: 'app/mailers/**/*.rb'
  allowed: 'project_relations, project'
  required: 'actionmailer'

documents:
  files: 'app/models/attack_document.rb, app/models/control_document.rb,
      app/models/document.rb, app/models/improvement_document.rb'
  required: 'activerecord'

admins:
  files: 'app/models/admin.rb, app/models/admin_project.rb,
      app/models/admin_project_area.rb'
  required: 'activerecord'
  allowed: 'project_relations, project'

consolidated_control:
  files: 'app/models/consolidated_area.rb, app/models/consolidated_checker.rb,
      app/models/consolidated_manager.rb, app/models/consolidated_project.rb'
  allowed: 'activerecord'

presenters:
  files: 'app/presenters/**/*.rb'

devise:
  gems: 'Devise'

chartdraw:
  files: 'lib/chart_helper.rb'
  allowed: 'project_relations'

actioncontroller:
  gems: 'ActionController'

activerecord:
  gems: 'ActiveRecord'

actionmailer:
  gems: 'ActionMailer'

accesscontrol:
  gems: 'CanCan'
```

Listing D.1: `PLC Attorneys` architectural specification file

D.2 Reflexion Model

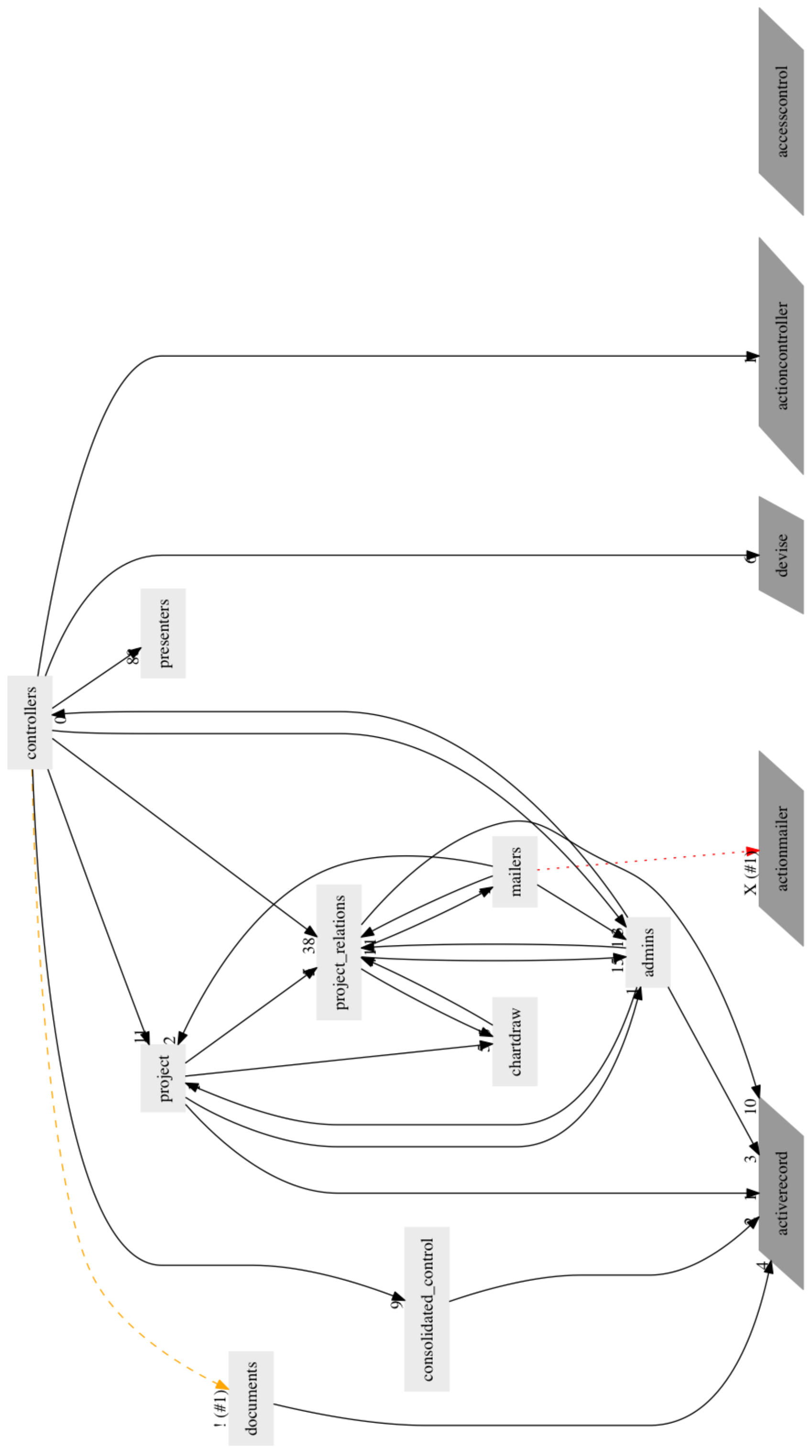


Figure D.1: RM automatically computed by `ArchRuby` for `PLC Attorneys`