An Empirical Study on the Impact of Deimplicitization on Comprehension in Programs Using Application Frameworks

Jürgen Cito*
TU Wien, Austria
MIT CSAIL
Cambridge, MA, USA

Jiasi Shen*
MIT CSAIL
Cambridge, MA, USA

Martin Rinard
MIT CSAIL
Cambridge, MA, USA

ABSTRACT

Background: Application frameworks, such as Ruby on Rails, introduce abstractions with the goal of simplifying development for particular application domains, such as web development. While experts enjoy increased productivity due to these abstractions, the flow of the programs is often hard to understand for non-experts and newcomers due to implicit flow and concealed lower level action that seems like "magic".

Objective: We conjecture that converting these implicit flows into an explicit and unified form can help non-experts comprehend the programs using these frameworks. We call the process of unifying distributed, implicit flows into a single routine *deimplicitization*.

Method: We want to conduct an experiment that studies the impact of deimplicitization on program comprehension. Particularly, we want to study how software developers with different expertise (novices/students, framework experts/professional developers) can answer comprehension questions differently with respect to time and correctness, under the treatments of either a deimplicitized version of the program in Python or the original version of the program in Ruby on Rails.

ACM Reference Format:
Jürgen Cito, Jiasi Shen, and Martin Rinard. 2020. An Empirical Study on the Impact of Deimplicitization on Comprehension in Programs Using Application Frameworks. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3379597.3387507

1 INTRODUCTION

Application frameworks, such as Ruby on Rails, introduce abstractions with the goal of simplifying development for particular application domains (e.g., web development). Proponents of these frameworks argue that the high learning curve introduced by the abstractions are justified because they result in increased productivity for expert developers. However, an unfortunate ancillary consequence is that programs written in these frameworks are often hard to understand for non-experts and newcomers to the codebase.

Continuing the example of Ruby on Rails, its abstractions introduce difficulties in the following aspects:

- Implicit flow of dynamic languages and dynamic dispatch.
- Intensive use of indirection through listeners.
- "Magic" functionality introduced through metaprogramming.[1]
- Behavior in the codebase without explicit flows due to abstractions in application frameworks (e.g., MVC — Model View Controller paradigm).

These implicit flows provide high-level programming abstractions for these frameworks. However, a downside is that programs using these frameworks may perform low-level actions not apparent in the source code, which often appear as unpredictable behavior or unexpected semantics [1, 2, 7–9] to non-experts.

1.1 Deimplicitization

We conjecture that converting these implicit flows into an explicit and unified form can help non-experts comprehend the programs using these frameworks. We call the process of unifying implicit flows into a single routine *deimplicitization*.

To deimplicitize programs that use these frameworks, we implemented an approach, Konure [6], that uses active learning to infer programs that access relational databases. The Konure paper presents experiments where Konure infers the functionality of programs in various languages (Ruby on Rails, Java, and Python) and regenerates the functionality in Python.

1.2 Konure

Figure 1 provides an overview of the Konure architecture. When Konure executes the program, the Konure proxy collects the traffic between the program and the database. As a result Konure observes all of the low-level SQL queries performed by the program. Note that these queries are often hidden from developers — the application frameworks are designed to implicitly perform low-level queries based on high-level source code.

Konure uses an internal domain specific language (DSL) to represent the observed program functionality. This DSL supports (a subset of) SQL queries, along with variable references and control structures such as conditional branches and bounded loops. Konure works with black box programs whose externally visible behavior conform to the DSL. Because Konure does not need to analyze the source code of these programs, the programs may be implemented in any languages or application frameworks. Given a potentially

*Both authors contributed equally to this research.

[1] https://medium.com/ruby-on-rails/demystifying-the-magic-of-rails-416d2195f098

MSR '20, October 5–6, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7517-7/20/05…$15.00
https://doi.org/10.1145/3379597.3387507

implicit program whose behavior conforms to the DSL, Konure infers the program functionality and translates it into a unified and explicit Python routine. When the original program is implemented with implicit application frameworks such as Ruby on Rails, the regenerated Python routine serves as a deimplicitized version of the original program.

This study investigates how deimplicitization affects the comprehension of programs that are built with application frameworks.

2 HYPOTHESES & RESEARCH QUESTIONS

We formulate our high level hypothesis as follows:

> *Deimplicitization can help developers comprehend implicit programs.*

In the text that follows, we refer to the deimplicitized version of an implicit program as its corresponding "deimplicitized program."

2.1 Research Questions

Our empirical study reflects on the following research questions:

- **RQ1:** Do deimplicitized programs reduce the *time* needed to complete comprehension tasks on the corresponding implicit programs?
- **RQ2:** Do deimplicitized programs increase the *correctness* of comprehension tasks on the corresponding implicit programs?
- **RQ3:** Does the participants' *expertise* in the application frameworks influence the time to complete or the correctness of the tasks?

These research questions follow from our high level hypothesis on deimplicitization.

2.2 Hypotheses

We translate the research questions **RQ1** and **RQ2** into the following testable hypotheses:

- $H1_0$: There is no difference in the response *time* between participants that access or do not access the deimplicitized programs.
- $H2_0$: There is no difference in the *correctness* of responses between participants that access or do not access the deimplicitized programs.

Their corresponding alternative hypotheses are as follows:

- $H1_a$: The response time is shorter for participants that access deimplicitized programs than participants that do not.
- $H2_a$: The correctness of responses is higher for participants that access deimplicitized programs than participants that do not.

To study **RQ3**, we will conduct two separate experiments:

- **Experiment I** focuses on novices.
- **Experiment II** focuses on experts.

For each experiment we will independently test the two hypotheses described above.

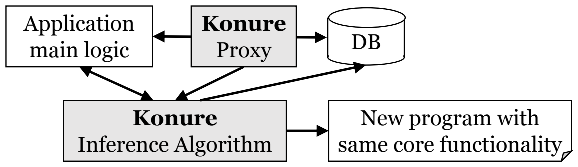
Application main logic
Konure Proxy
DB
Konure Inference Algorithm
New program with same core functionality

Figure 1: The Konure architecture, including a transparent proxy interposed between the application and the database to observe the generated database traffic.

3 VARIABLES

We briefly describe the variables involved in our experiments. The independent variable is:

- **Deimplicitization**, dichotomous variable. This variable represents whether a participant has access or has no access to a deimplicitized program.

The dependent variables can be described as follows:

- **Time Spent**, in seconds ($H1_0$). This variable represents the time from a participant being presented with a task until providing an answer.
- **Correctness**, dichotomous variable ($H2_0$). This variable represents whether a participant answers a task correctly or incorrectly. We only distinguish between fully correct answers. There is no "partial credit" for answers.

We additionally introduce the following control variables:

- **RoR Familiarity**, in years. This variable represents a participant's prior experience working with Ruby on Rails, which is the application framework of the implicit programs.
- **Python Familiarity**, in years. This variable represents a participant's prior experience working with Python, which is the language of the deimplicitized programs.
- **SQL Familiarity**, dichotomous variable. This variable represents whether a participant has prior experience with SQL, either through direct working experience or through education.
- **Study Object Difficulty/Complexity**. This variable represents the level of difficulty/complexity of a study object, which is an open source Ruby on Rails project supported by Konure.

We anticipate that the major confounding variable is the prior experience with the relevant application framework and languages, that is, Ruby on Rails, Python, and SQL. We plan to control for the effect of these confounding variables by focusing the population for each experiment on a small range of expertise levels.

```
def get_admin_pages_id (conn, inputs):
  util.clear_warnings()
  outputs = []
  s0 = util.do_sql(conn, "SELECT  `pages`.* FROM `pages
      ` WHERE `pages`.`id` = :x0 LIMIT 1", {'x0':
      inputs[0]})
  outputs.extend(util.get_data(s0, 'pages', 'id'))
  outputs.extend(util.get_data(s0, 'pages', 'title'))
  outputs.extend(util.get_data(s0, 'pages', 'slug'))
  outputs.extend(util.get_data(s0, 'pages', 'body'))
  return util.add_warnings(outputs)
```

Figure 2: A Python routine generated by Konure after inferring an API of a Ruby on Rails application.

4 MATERIALS & OBJECTS

As part of our study objects we use open source applications of various sizes and complexities that are built with application frameworks. Specifically, we use the Ruby on Rails projects in the experiments of the Konure paper [6]:

- **Enki**, Content Management. The source code is available at https://github.com/xaviershay/enki.
- **Fulcrum**, Agile Project Management.
  The source code is available at
  https://github.com/fulcrum-agile/fulcrum.
- **Kandan**, Online Chat. The source code is available at https://github.com/kandanapp/kandan.
- **Blog**, Sample Application. The source code is available at https://guides.rubyonrails.org/getting_started.html.

The Enki, Fulcrum, and Kandan applications are open source projects each with hundreds of stars on GitHub. The Blog application is adapted from the standard online tutorial.

Each of these applications consist of a user interface as a web page and a back-end server that accepts API calls from the web browser. When an API is called, the application server translates the input parameters into SQL queries against a relational database and returns results extracted from the results of the queries. The web interface then renders the query results in user-friendly layout and displays the resulting web page.

From these applications we adopt APIs that are studied in the Konure paper [6]. For each API, we obtain its deimplicitized Python routine by adapting from the code regenerated by Konure.[2] As an example, Konure regenerated the Python routine in Figure 2 for an API in Enki that retrieves a page by its ID.

As a preprocessing step, we will extend the procedure to systematically rewrite the Python routines to eliminate unnecessary artifacts that were automatically generated by Konure, including:

- eliminating redundant or unused statements,
- removing redundant conditional checks if both of the conditional branches are equivalent,
- replacing systematic variable names with descriptive words, such as the names of relevant tables, and
- renaming library function calls, data fields, input arguments, and output operations, so that the code is more readable.

[2]The code regenerated by Konure is available at an online appendix associated with the paper: http://people.csail.mit.edu/jiasi/pldi2019.code/.

The goal is to allow the deimplicitized code to provide similar information as in potential deployment scenarios.

5 PARTICIPANTS/SUBJECTS

For both experiments, we recruit participants with software development experience who have at least one year of Python programming experience and are familiar with relational databases and SQL syntax. Because Python is widely used and because databases are often taught in undergraduate institutions, we believe that this population is broad enough for general interest. Additionally:

- **Experiment I** participants must have no prior experience with Ruby on Rails (*novices*).
- **Experiment II** participants must have at least one year of programming experience with Ruby on Rails (*experts*).

Before starting each experiment, we plan to first conduct a pilot study to estimate the effect sizes. Based on this preliminary data, we will perform a power analysis and aim for at least 40% statistical power, which is common for software engineering experiments [4], at the standard 0.05 level ($\alpha = 0.05$). We will then determine the sample size based on the results of the power analysis and the availability of participants.

6 EXECUTION PLAN

For each experiment, we assign participants into two groups:

- **Control Group** participants are shown only the implicit programs, i.e., the Ruby on Rails source code.
- **Treatment Group** participants are shown both the implicit programs and their corresponding deimplicitized programs, i.e., both the Ruby on Rails source code and the corresponding deimplicitized Python routine.

To familiarize participants with the study environment, we plan to start the session with a warm up task to explain the setting. The code is shown in a syntax highlighted text editor. We do not allow participants to execute or modify the code.

Participants are then asked to comprehend several application APIs. For each API, participants answer questions about the functionality shown in the code. As soon as the code and question are displayed, we automatically start to measure the time to task completion, which is completed as soon as participants move on to the next question (or complete the experiment session). We follow up with open-ended questions for participants to discuss their experience.

7 ANALYSIS PLAN

To provide an overview of the study results, we plan to present summary statistics (mean, median, standard deviation, max, min) over the treatment groups and overall for the dependent variables (time in seconds, number of correctly fulfilled tasks). Additionally, we want to show the distribution as part of a histogram.

As for significance tests, we plan to follow a frequentist approach. We would first test for normality by conducting a Shapiro-Wilk test. Depending on the distribution, we would then either perform a t-test or Mann-Whitney U test. We will do this for our variables **Time Spent** and **Correctness**. Since correctness is a binary variable, our measure will be the number of correct answers given.

We initially calculated effect size as part of our power analysis. We also set out to estimate the extent of how substantially different the measures in our full dataset are by calculating the effect size again. We choose the particular procedure to calculate effect size given the distribution of our sample measures. Cohen's d is an appropriate comparison between two means, usually to accompany a t-test [3]. Cliff's delta is a non-parametric effect size measure that is used to calculate the frequency of values in one distribution differ from values in another distribution [5].

8 THREATS TO VALIDITY

Prior experience with Ruby on Rails could affect the results. We mitigate this threat by focusing each experiment on only novices or only experts.

Using a specific sample of small to medium size Ruby on Rails applications and APIs could limit the ability to generalize the results, especially for **RQ1** (time), to larger applications or applications with different code quality (e.g., availability of comments). We anticipate that the results for **RQ2** (correctness) and **RQ3** (expertise) are still relevant for larger applications, given that the focus is on one input/output flow targeting a particular entry point of the application.

Using a research tool (Konure) could limit the ability to generalize the results to deployment scenarios. We mitigate this threat by preprocessing the Konure deimplicitization outputs systematically to mimic potential deployment scenarios.

Focusing on only one abstraction framework (Ruby on Rails) could limit the ability to generalize the results to other abstraction frameworks, especially frameworks that adopt different abstraction strategies. However, the particular abstractions (model abstraction, view abstraction, routing, inversion of control) that we target with Konure are traits that are prevalent in many other application frameworks, such as Python/Django, PHP/Symfony, and Java/Spring. This increases the likelihood that our results generalize within the scope of these application frameworks.

Using only one deimplicitization tool could limit the ability to generalize the results to other potential tools that deimplicitize programs in other domains of computation.

ACKNOWLEDGEMENT

The first author was funded by the Swiss National Science Foundation (SNSF) under project Automated Synthesis and Repair of Infrastructure Code (no. 178036).

REFERENCES

[1] T. Chen, W. Shang, Z. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. *IEEE Transactions on Software Engineering*, 42(12):1148–1161, dec 2016.

[2] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 3–14, New York, NY, USA, 2013. ACM.

[3] Jacob Cohen. *Statistical power analysis for the behavioral sciences.* Routledge, 2013.

[4] T. Dybå, V. B. Kampenes, and D. I.K. Sjøberg. A systematic review of statistical power in software engineering experiments. *Information and Software Technology*, 48(8):745 – 755, 2006.

[5] Guillermo Macbeth, Eugenia Razumiejczyk, and Rubén Daniel Ledesma. Cliff's delta calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica*, 10(2):545–555, 2011.

[6] Jiasi Shen and Martin C Rinard. Using active learning to synthesize models of applications that access databases. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 269–285. ACM, 2019.

[7] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. Understanding database performance inefficiencies in real-world web applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, CIKM '17, pages 1299–1308, New York, NY, USA, 2017. ACM.

[8] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. How not to structure your database-backed web applications: A study of performance bugs in the wild. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 800–810, New York, NY, USA, 2018. ACM.

[9] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. Powerstation: Automatically detecting and fixing inefficiencies of database-backed web applications in ide. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 884–887, New York, NY, USA, 2018. ACM.