Achilleas Achilleos
Lidia Fuentes
George Angelos Papadopoulos (Eds.)

LNCS 14614

Reuse and Software Quality

21st International Conference
on Software and Systems Reuse, ICSR 2024
Limassol, Cyprus, June 19–20, 2024
Proceedings

Springer

Lecture Notes in Computer Science **14614**

The series Lecture Notes in Computer Science (LNCS), including its subseries Lecture Notes in Artificial Intelligence (LNAI) and Lecture Notes in Bioinformatics (LNBI), has established itself as a medium for the publication of new developments in computer science and information technology research, teaching, and education.

LNCS enjoys close cooperation with the computer science R & D community, the series counts many renowned academics among its volume editors and paper authors, and collaborates with prestigious societies. Its mission is to serve this international community by providing an invaluable service, mainly focused on the publication of conference and workshop proceedings and postproceedings. LNCS commenced publication in 1973.

Achilleas Achilleos · Lidia Fuentes ·
George Angelos Papadopoulos
Editors

Reuse and Software Quality

21st International Conference
on Software and Systems Reuse, ICSR 2024
Limassol, Cyprus, June 19–20, 2024
Proceedings

Springer

Editors
Achilleas Achilleos
Frederick University
Nicosia, Cyprus

Lidia Fuentes
University of Malaga
Malaga, Spain

George Angelos Papadopoulos
University of Cyprus
Nicosia, Cyprus

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-031-66458-8 ISBN 978-3-031-66459-5 (eBook)
https://doi.org/10.1007/978-3-031-66459-5

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

This volume contains the proceedings of the 21st edition of the International Conference on Software and Systems Reuse (ICSR 2024). The conference was organized in the beautiful city of Limassol, Cyprus during June 19–20, 2024.

ICSR is the premier international event in the software reuse community and its main goal is to present the most recent advances in the area of software and systems reuse and to promote an exchange of ideas and best practices among researchers and practitioners.

The 21st edition's main theme was "Sustainable Software Reuse", promoting the sustainable reuse of robust, validated components and libraries. This is beneficial for the software community as it allows for the rapid development of high-quality applications with a reduced environmental footprint. Additional innovative targets of software development and reuse are IoT, cloud/edge computing or Artificial Intelligence. Recent advances focused on accelerating development processes but also opened new possibilities for creating innovative and customized solutions. We solicited papers that explored these, and every aspect of software reuse. This edition featured sessions on Artificial Intelligence and Reuse, Variability and Reuse, Adaptation and Reuse and Code Reuse which presented results that emphasize methods, techniques and tools for sustainable software reuse.

Following the current practices in reviewing, we used a double-blind process. We received a total of 25 submissions divided into two main tracks: Research and Late Breaking Results tracks. Each paper was reviewed by three Program Committee members followed by a discussion through Easy Chair, with the involvement of the General and Program Chairs. The Research track received 20 abstracts and 19 full papers were reviewed, out of which seven were accepted as full papers and two as short papers. The full paper acceptance rate of the Research track was 35% and the overall acceptance rate of all tracks was 44%.

We were delighted to have two keynote speakers. Hermann Kaindl delivered a keynote on "Many Flavors of Reuse and Reusability" and Georgia Kapitsaki delivered a keynote on "Open-source software issues practitioners face: general issues and licensing concerns".

We are grateful for the hard work of the Program Committee which helped to assure a high-quality conference. Furthermore, we want to thank the Organizing Team and the Steering Committee for their support in ensuring a successful conference.

June 2024

Achilleas Achilleos
Lidia Fuentes
George Angelos Papadopoulos

Organization

General Chair

George A. Papadopoulos	University of Cyprus, Cyprus

Program Committee Chairs

Achilleas Achilleos	Frederick University, Cyprus
Lidia Fuentes	University of Málaga, Spain

Steering Committee

Eduardo Almeida	Federal University of Bahia, Brazil
Goetz Botterweck	Lero, Trinity College Dublin, Ireland
Rafael Capilla	Rey Juan Carlos University, Spain
John Favaro	Trust-IT, Italy
William B. Frakes	IEEE TCSE Committee on Software Reuse, USA
Martin L. Griss	Carnegie Mellon University, USA
Oliver Hummel	Mannheim University of Applied Sciences, Germany
Hafedh Mili	Université du Québec à Montréal, Canada
Nan Niu	University of Cincinnati, USA
George A. Papadopoulos	University of Cyprus, Cyprus
Claudia M. L. Werner	Federal University of Rio de Janeiro, Brazil

Program Committee

Mercedes Amor	Universidad de Málaga, Spain
Apostolos Ampatzoglou	University of Macedonia, Greece
Claudia P. Ayala	Universitat Politècnica de Catalunya, Spain
Olivier Barais	IRISA/Inria/Univ. Rennes l, France
Sihem Ben Sassi	University of La Manouba, Tunisia
Mireille Blay-Fornarino	Université Côte d'Azur, France
Jan Bosch	Chalmers University of Technology, Sweden
Rafael Capilla	Universidad Rey Juan Carlos, Spain

Stephanie Challita	Inria, France
Alexander Chatzigeorgiou	University of Macedonia, Greece
Marsha Chechik	University of Toronto, Canada
Serge Demeyer	Universiteit Antwerpen, Belgium
Chizlane El Boussaidi	École de Technologie Supérieure, Canada
Alexander Felfernig	TU Graz, Austria
Jessie Galasso	McGill University, Canada
José A. Galindo	University of Seville, Spain
Barbara Gallina	Mälardalen University, Sweden
Iris Groher	Johannes Kepler University Linz, Austria
José Miguel Horcas Aguilera	Universidad de Málaga, Spain
Nan Niu	University of Cincinnati, USA
Mourad Oussalah	LINA Laboratory, University of Nantes, France
Gilles Perrouin	University of Namur, Belgium
Cordana Rakic	University of Novi Sad, Serbia
Klaus Schmid	University of Hildesheim, Germany
Abdelhak Seriai	University of Montpellier, France
Quentin Stiévenart	Université du Québec à Montréal, Canada
Xhevahire Térnava	Université de Rennes 1, France
Zhipeng Xue	National University of Defense Technology, China
Wei Zhang	Peking University, China
Christina von Flach	Federal University of Bahia, Brazil

Abstracts of the Invited Talks

Many Flavors of Reuse and Reusability

Hermann Kaindl

TU Wien and Vienna University of Economics and Business, Vienna, Austria
hermann.kaindl@tuwien.ac.at

Abstract. In this keynote, I reflect on a few decades of my own encounters and experiences with many "flavors" of software reuse and reusability (RR), both in industry and in scientific research. This includes RR of program code, software design and architecture, as well as RR based on business models, domain models and ontologies. I will also talk about feature models/product line engineering (PLE) and case-based reasoning (CBR), and in particular their integration. I will try to generalize and provide lessons learned that go beyond my specific experiences. Some of these approaches were based on and are related to Artificial Intelligence (AI), where I was heavily involved in research in the 1980s and 1990s. Given the recent and current successes of AI, I will finish with a few speculations on future RR by making use of AI.

Open Source Software Issues Practitioners Face: General Issues and Licensing Concerns

Georgia Kapitsaki

Department of Computer Science, University of Cyprus, Nicosia, Cyprus
gkapi@ucy.ac.cy

Abstract. Open source software is used by many practitioners, whereas many organizations choose to reuse open source software or make their products open source. At the same time, Stack Overflow and other Stack Exchange sites are widely used by practitioners in order to get help in programming and other issues they face by fellow practitioners. This talk will cover issues practitioners face when it comes to open source software, whereas special emphasis will be put on the area of licensing of open source software and the relevant issues in that specific area. In the first part of the talk, open source software licensing will be briefly presented. In the second part, analysis performed by our research group on Stack Exchange sites concerning open source software issues in general and licensing more in more specific will be covered. The talk will also explore important parameters, when choosing an open source software license for a project and will present resources and techniques that can assist in this process.

Contents

Code Reuse

Artificial Intelligence and Reuse

The Unexpected Blocking of Code Understanding in AI-Based Code Summarization: Observations and Concerns from a Study on Cross-Project Learning Performance

Sixuan Zhang and Yan Liu(✉)

School of Software Engineering, Tongji University, Shanghai, China
{2231514,yanliu.sse}@tongji.edu.cn

Abstract. Code comments are one of the pillars of software reuse, as they assist program comprehension activities and reduce maintenance costs. Code comments are prevalent with missing, outdated, and mismatched issues, whereas code summarization could help by generating concise comments. With the development of deep learning, the emergence of pre-trained models has brought notable progress in code summarization. However, these Transformer-based models prefer literal representations of code and may struggle to utilize structural features, which may affect the models. Structured features are common to codes; real code is always in a project. Therefore, in this work, we evaluate the models by the observations of the models' cross-project generalization performance, including metrics, tendency observations, and iterative observations. The results show that current models perform surprisingly poorly, with BLEU scores over 50% lower, while ROUGE and METEOR scores are over 30% lower on cross-project test sets. Current models may rely more on project-specific than generic code features to understand code. We link this to the model's performance on different code channels. It is suggested that the model performance from different code channels be evaluated and the implicit code channels further explored.

Keywords: Code Summarization · Pre-trained Models · Cross-Project Generalization

1 Introduction

Code summary (or code comment) is the natural language description of source code. Code summaries can be written for many purposes, and those explaining the logic and functionality of the code are one of the most common [1]. When it comes to software reuse, code comments are crucial: they improve the code's readability and thus its reusability. During the software life cycle, code maintenance is one of the most costly and time-consuming processes [2]. Concise

A. Achilleos et al. (Eds.): ICSR 2024, LNCS 14614, pp. 3–17, 2024.
https://doi.org/10.1007/978-3-031-66459-5_1

code summaries enable developers to quickly and accurately understand the code's structure and purpose without delving into its implementation details.

However, in real projects, code summaries have common quality issues such as missing, outdated, and mismatched [3]. Even well-known open source organizations have much serious code summary lacks [4]. An automated code summarization method can help to solve these problems by generating high-quality code summaries.

Current code summarization mainly uses deep learning-based methods. In the beginning, the Neural Machine Translation (NMT) models are migrated to learn the code and generate code summaries [5], which treats the code as natural language and ignores the structural information of the code. Therefore, researchers started to use Abstract Syntax Tree (AST) as support to enable the models to learn the structural information of the code [6].

In recent years, with the occurrence of pre-trained models, many studies have started to use pre-trained models based on the Transformer [7] architecture and fine-tuned these models on code summarization tasks [8,9]. The pre-trained models have made remarkable progress in code summarization tasks. However, these Transformer-based [7] pre-trained models take the literal code representation as input and cannot learn the structural information of the code directly. Some attempts to utilize AST have also made little progress [9,10]. Structural information seems less important with the notable metrics scores improvement achieved by the pre-trained models [9,11]. Still, since the code contains rich structural information, it should not be treated as texts [12]. Therefore, we make observations on how this contradiction would affect the model.

The learnable information within the code is manifested as code features, some shared in various functions, such as structural features containing structural information. In code summarization, strict classification and evaluation methods for the model's ability to learn code features are currently absent. Since the structural information of the code is common to the code, the model's generalization ability may be worse as the structural information is lacking. Meanwhile, since real code is often in a project, we evaluate model performance by the cross-project generalization ability of the model. We conduct metrics evaluation, tendency observation, and iterative observation experiments. We divide the filtered Funcom [13] dataset by projects and randomly sample projects to form a training set, an internal test set (ITS), and an external test set (ETS), where the ITS is from the same set of projects as the training set.

The results show that the performance of the models on ETS is surprisingly poor. These pre-trained models consume lots of resources to learn the code to extract generic code features but may not exhibit strong cross-project generalization abilities. They may lack understanding of functions' internal logic and instead rely on project-specific features (e.g., programming style) to improve the metrics scores. From the perspective of code naturalness [12,14], we link this to a difference in the performance of the model on the code channels [12]: the models perform well on the explanatory channeler, but fail to catch up on the

algorithmic channel. Observing the model's performance from the perspective of different code channels is suggested.

The main contributions of this work are as follows:

- A cross-project experiment to evaluate models' code comprehension ability.
- Further analysis and consideration of the model's performance.

The remainder of this paper is organized as follows. In Sect. 2, we review the related works on code summarization. In Sect. 3, we describe our research methodology, research questions, and experimental design. In Sect. 4, we summarize and analyze the experimental results, showing our observations and concerns. In Sect. 5, we discuss our work and possible future directions. In Sect. 6, we present threats to the validity of this work. In Sect. 7, we conclude the paper.

2 Related Work

2.1 Code Summarization

Code summarization methods are mainly categorized into information retrieval (IR) based and deep learning (DL) based [15]. With the development of deep learning, DL-based methods have become the mainstream. Initial research treated code as a special natural language, using Recurrent Neural Networks (RNN)-based [5] and Transformer-based [8,16] models to generate code summaries. In addition to literal information, code also contains rich structural information. The abstract syntax tree is the most widely utilized structural information of codes [15]. There are various ways to utilize it: for models, Leclair et al. [6] and Zhou et al. [17] use Graph Neural Networks (GNNs) and their variants to learn AST as additional inputs to the model; Liang et al. [18] design a novel RNN to learn AST. For the form of the AST, besides retaining the tree structure of the AST, some studies have tried to use traversed AST as model inputs, such as Structure-based Traversal (SBT) proposed by Hu et al. [19]. However, Ahmad et al. [8] demonstrate that direct input of SBT into the Transformer model causes additional cost but does not improve the model's performance.

Pre-trained code models are trained on large datasets and fine-tuned at downstream tasks. In code summarization, the pre-trained models have achieved notable performance improvement [9,20]. However, these Transformer-based pre-trained models prefer literal code representations and are not structure-friendly. Recent pre-trained models have either learned only code sequences [21] or utilize the AST without performance improvement [9,10]. CodeBERT [9] and UnixCoder [10] attempt to use AST as additional model inputs but fail to improve the performance. CodeT5 [20] tags identifiers in the input sequence, utilizing only a small fraction of the information from AST. Therefore, in this work, we select the latest pre-trained models [9–11,20,21] that have been fine-tuned in the code summarization task and observe the code comprehension ability of these models.

2.2 Evaluating Code Summarization

The output of the code summarization task is comments of the target code. Hence, the evaluation of the code summarization model focuses on measuring the similarity of the generated comments to the reference comments. Evaluation methods are mainly categorized into n-gram-based and embedding-based [22]. The n-gram-based methods measure the word overlap of the model predictions with the reference, including BLEU [23], Rouge [24], and METEOR [25], etc. Embedding-based methods compute the semantic similarity of the model's prediction to the reference, including word-based similarity [26] and sentence-based similarity [27]. These methods generate vectorized representations of two sentences and then measure their cosine similarity or Euclidean similarity. In this work, we use the more commonly used n-gram based approach to evaluate models to align with existing works and avoid unexpected bias. We observe the models' ability to comprehend the code through their metric scores of the predictions. In addition to using automated metrics, we supplement with tendency and iterative observations.

3 Methodology

3.1 Background and Research Questions

In addition to literal information, code is rich in structural information. Rahman et al. [12] state that the structural information should be considered to represent code better. AST is a vital way to express the structure information of the source code [15]. However, current pre-trained models are Transformer-based [7], which prefer literal representations of code while lacking the direct utilization of AST. Our motivation comes from this contradiction between models and code. Therefore, we formulate our first research question:

RQ1. Given that a transformer-based pre-trained solution is the most promising architecture for current code-learning models, is it possible to observe the code comprehension capabilities of those models with a meticulous experiment?
To answer RQ1, we need to review the pre-training and evaluation methods of the models. In the code summarization task, the models learn the code and generate its comments. Therefore, the quality of the generated comments could reflect the model's code comprehension capabilities. To learn generic code features, pre-trained models are often trained on a large corpus containing real projects and codes. The structural information of the code is one of the code features shared by the code in the corpus. In addition, code in real-world projects unavoidably incorporates project-specific features such as programming specifications, domain knowledge, and even programming styles, which the model may also learn. Therefore, evaluating models on projects that have not been trained could observe the model's ability to learn the generic code features.

Answer to RQ1: As the models are pre-trained on code from real-world projects, they may acquire both project-agnostic and project-specific features.

The models' comprehension capability can be assessed by observing its performance across projects, i.e., comparing its performance on projects present in the training set to those absent from it.
From this, we can formulate our second research question:
RQ2. With the experimental design proposed in this paper, are there any findings that expose the underestimated challenges of understanding code for code summarization models? If so, to what extent may specific concerns be addressed to enhance the architecture of transformer-based pre-trained models?
Despite the notable progress made by pre-trained models in the code summarization task, the lack of structural information contained in the generic code features may still affect the model's generalization ability. We accordingly formulate a subquestion for RQ2:

RQ2.1. With an aligned scale, how well do the models perform in cross-project generalization?
As mentioned, we conducted experiments on pre-trained models at aligned scales to avoid unexpected bias. Also, since the model may also learn project-specific information, we formulate the second subquestion:

RQ2.2. Does the model, as expected, understand code from both generic and project-specific perspectives relatively balanced by learning with pre-training for code summarization? If not, are any de facto preferences exhibited inside the current experiment scope, and to what extent do they influence the model performance?
An ideal model for code summarization should understand the code in a balanced way from generic and project-specific features. However, it is difficult to directly observe whether the model understands the code as balanced. To answer RQ2 and its subquestions, we design a cross-project experiment. We divide the dataset into two project sets, one of which is further divided into a training set, a test set as ITS, and the other as ETS. We believe that the ability and the way the model understands the code could be assessed by observing the differences in the model's performance on the datasets. We perform full-parameter fine-tuning on the training set and evaluate the model on the ITS and ETS. Our evaluation methods include metrics evaluation, tendency observations, and iterative observations.

3.2 Design of the Experimental Study

Our design of the experimental study is shown in Fig. 1:

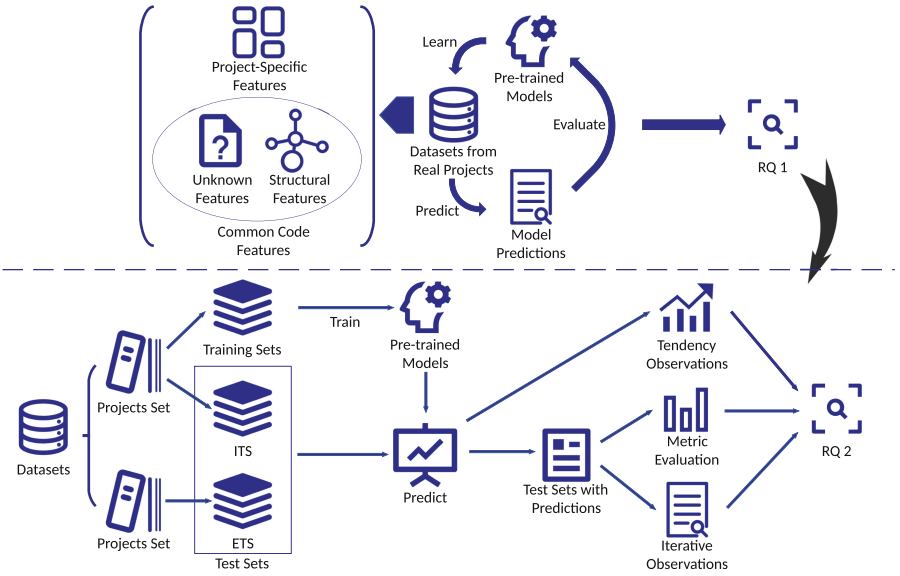

Fig. 1. Design of the experimental study

3.3 Experimental Setup

Pre-trained Models. Table 1 lists the pre-trained models chosen for our experiments. These are the latest Transformer-based models in recent years and have been fine-tuned in code summarization tasks. We also select versions of these models with relatively similar parameters for alignment.

Table 1. The chosen pre-trained models

Model	Year	Parameters
CodeBERT [9]	2020	125M
UniXCoder [10]	2022	125M
PLBART [21]	2021	140M
CodeT5 [20]	2021	220M
CodeT5s [20]	2021	60M
CodeT5P [11]	2023	220M

We get the pre-trained models on HuggingFace[1] and perform full-parameter fine-tuning for the code summarization task.

[1] https://huggingface.co/.

Dataset and Training. We choose Funcom [13] as our origin dataset, a dataset of 2.1 million Java functions and their comments. Each function is identified by its unique ID and its project ID.

To explore the model's performance on cross-project functions, we divide the dataset by projects and select projects with more than 3,000 functions to ensure it is sufficient for the training and evaluation. We have chosen 19 projects from Funcom as our dataset, comprising 198,000 functions with internal comments removed.

To ensure that the experimental results are not affected by particular projects, we divide the dataset before the start of each training: first, we randomly divide the projects into training-testing projects and testing-only projects, and then randomly sample about 80% of the functions in each training-testing project as the training set, and the rest as the ITS. All functions in the test-only projects are used as the ETS. The model is fine-tuned on the training set and evaluated separately on the ITS and ETS. We also try to ensure that the ITS and ETS are similar in size in each division.

3.4 Evaluation Methods

Metrics. We use BLEU-4 [23], ROUGE-L F1 [24], and METEOR [25], which are widely used metrics in the code summarization tasks, to evaluate the models' performance on ITS and ETS.

According to Roy et al. [28], for the code summarization task, systematic improvements in comments quality are not guaranteed when the metric score improvement is below a particular value, while they also observe statistically significant performance differences between models with metric scores greater than 10 points. We believe that although the model's cross-project performance will be worse, the model, which has been pre-trained on a large amount of code, is capable enough to finish the cross-project code summarization task. Therefore, the model's ability remains acceptable, and its metric score difference may not exceed 5 points.

Tendency Observations. In the selected models, PLBART only learns code sequences, while CodeT5 utilizes the type information in the AST. Moreover, they have similar parameters. Therefore, they are chosen as representatives to observe the tendency of the model's performance on the ITS and ETS by inserting extra evaluation processes at every certain step in their training processes. We also compute the Spearman correlations between the model's metric scores on ITS and ETS. We believe the lack of some generic features will result in a slower improvement in the model's performance while converging more quickly on the ETS.

We also search for similar functions in ITS and ETS but not in the same project and compute the similarity of the model predictions. We use SentenceBERT [27] to convert code and comments into vectors and use cosine similarity as the reference of similarity. We first search for pairs of functions

with similarity close to 1, then gradually decrease the similarity target and observe the tendency of the prediction's similarity. If the model generates dissimilar comments for similar functions, it may indicate that it can capture some project-specific information that was not noticed.

Iterative Observation. We conduct similarity searches by functions, reference comments, and model predictions to identify similar entries in the dataset and iteratively observe the search results to supplement the tendency observations.

4 Observations and Concerns

We conduct three train-evaluate rounds and randomly divide the dataset before each training, and we only show the average values on the observations.

4.1 Metrics

The models' metric scores on ITS and ETS are shown in Table 2. All the models perform surprisingly poorly on the ETS. All the models have score gaps of more than 10 points on all metrics.

Table 2. Average metric scores on ITS and ETS

Model	BLEU-4			ROUGE-L F1			METEOR		
	ITS	ETS	DIFF	ITS	ETS	DIFF	ITS	ETS	DIFF
CodeBERT	42.77	11.69	−72.64%	64.12	39.23	−38.80%	62.86	38.18	−39.24%
UniXCoder	37.46	12.55	−66.51%	60.02	40.00	−33.36%	58.16	38.17	−34.37%
PLBART	42.46	19.73	−52.96%	58.96	39.07	−33.62%	53.46	31.69	−40.63%
CodeT5	44.34	22.13	−50.02%	61.00	42.15	−30.92%	55.43	34.85	−37.15%
CodeT5s	37.76	21.90	−41.90%	56.15	42.33	−24.62%	49.53	34.63	−30.09%
CodeT5P	46.64	22.13	−52.50%	62.63	41.93	−33.06%	57.46	35.12	−38.91%

Table 3. Correlations and p-values of changes in metric scores of the model on ITS and ETS

	BLEU	ROUGE
CodeT5	0.983^{**}	0.994^{**}
	< 0.001	< 0.001
PLBART	0.952^{**}	0.976^{**}
	< 0.001	< 0.001

4.2 Tendency Observations

The tendency observation of the metric scores is shown in Fig. 2, and correlation computations between scores are shown in Table 3. The results showed positive correlations between the models' scores on ITS and ETS, suggesting that the model's performance on the ETS is unlikely due to overfitting on the ITS. Thus, combining the results of the metrics evaluation, we can conclude our answer to RQ2.1:

Answer to RQ2.1: The models' ability to generalize across projects is surprisingly poor, with BLEU scores decreasing by over 50%, ROUGE and METEOR scores decreasing by over 30% for projects that are not in the training set. The models' performance on ETS exhibits a significantly positive correlation with its performance on ITS.

Table 4. Correlations and p-values between function similarity and model prediction similarity

	CodeBERT	UniXCoder	PLBART	CodeT5	CodeT5s	CodeT5P
Function	0.958^{**}	0.972^{**}	0.970^{**}	0.978^{**}	0.978^{**}	0.967^{**}
	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001

The tendency observation of the cosine similarity is shown in Fig. 3, and the correlation computations between the similarities are shown in Table 4. When the cosine similarity threshold of the functions is set to 1 (i.e., to find the most similar function with a similarity of no more than 1 for each function), the similarity of the model predictions is mostly greater than 0.9. As the similarity threshold decreases, the difference between the similarity of the model predictions and the function similarity slowly increases. When the function cosine similarity threshold is set to 0.6, the difference approaches 0.25. Thus, we can conclude our answer to RQ2.2:

Answer to RQ2.2: Within the scope of the current experiments, the model may prefer project-specific features to generic code features to understand code. Even for similar functions in different projects, the model may prefer to utilize the clues in their respective projects to generate summaries unless there is a high similarity. This bias may considerably affect the model's generalization ability.

4.3 Iterative Observation

After iterative observations of the search results, we draw some conclusions about the model and the dataset as follows:

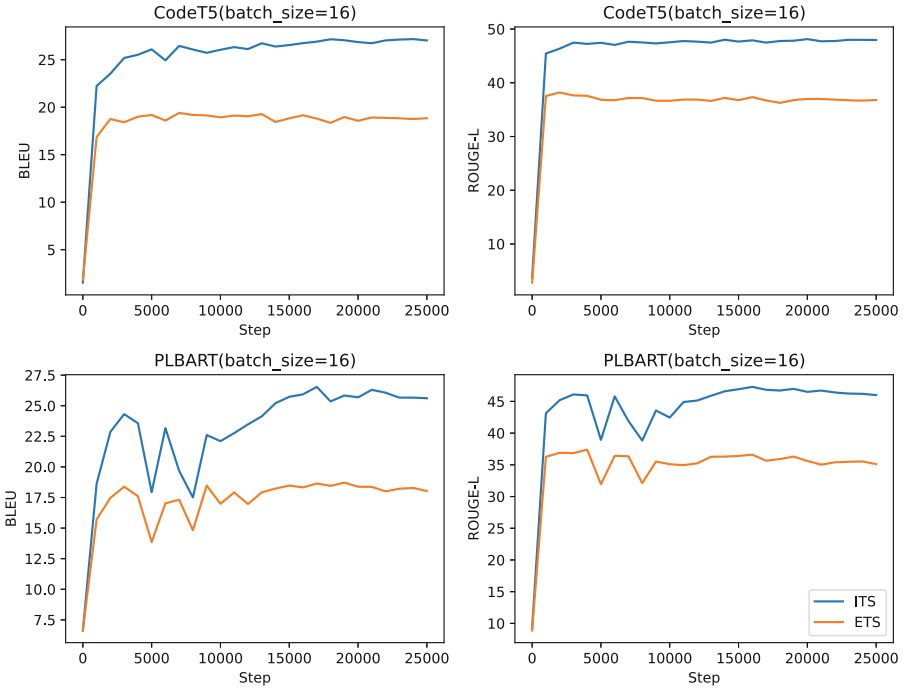

Fig. 2. The growth tendency of the model's metrics scores on ITS and ETS

Function Signature. For short functions, the model tends to extract information mainly from the function signature and lacks attention to the internal logic. As shown below.

```
// ProjectID: 22152, FunctionID: 22625727, ITS
// Ref: throws unsupported operation exception
// Pre: mark this entry as persisted
public Object setValue(Object val) {
    throw new UnsupportedOperationException("setValue operation not
        supported");
}


// ProjectID: 11412, FunctionID: 10363622, ETS
// Ref: throws unsupported operation exception
// Pre: returns the dependency manager for this connection
public DependencyManager getDependencyManager() {
    throw new UnsupportedOperationException("Dependency manager is
        not accessible by way of a remote connection");
}
```

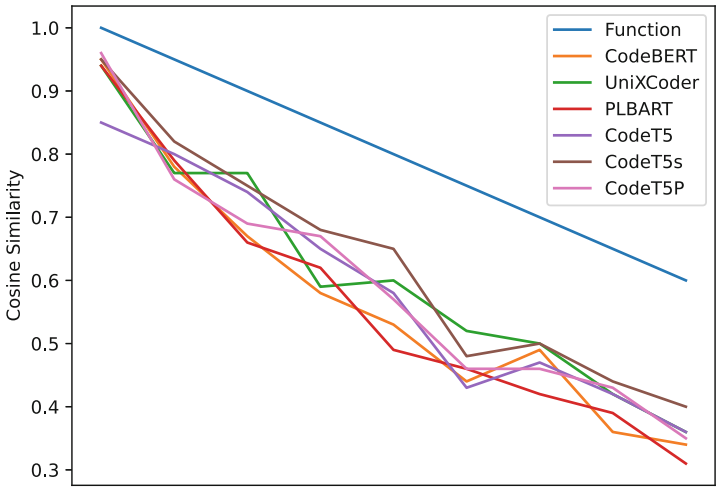

Fig. 3. The tendency of cosine similarity

Programming Specifications/Styles. There are lots of programming specifications/styles included in the dataset. For example, 2,613 function comments start with "setter for", whereas project 35895 alone contains 2,292; 2,053 function comments begin with "getter for", whereas project 35985 alone contains 1,714. Other obvious programming specifications/styles are "test(s) if...", "return(s) true if..." etc. The model can easily learn these programming norms/styles as follows:

```
// ProjectID: 6687, FunctionID: 5861913, ITS
// Ref: sets the maximal allowed size
// Pre: sets the maximum size of the file
public void setMaxSize(int maxsize) {
    _maxsize = maxsize;
}
```

```
// ProjectID: 11756, FunctionID: 12190528, ETS
// Ref: set the maximum message size
// Pre: setter for property max mmsize
public void setMaxMMSize(Integer maxMMSize) {
    this.maxMMSize = maxMMSize;
}
```

Interface Implementation. There are also lots of interface implementations in the dataset that have almost identical function signatures but very different internal logic. These functions are widely distributed across projects, as shown

below. It is worth noting that some interface implementations have the same comments, and the models are very likely to generate the same comments, which could be mistaken for data leakage. Also, considering the possible lack of utilization of the functions' internal logic by current models, these functions may be what the models really need to learn.

```
// ProjectID: 21945, FunctionID: 22358108, ITS
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == addComment) {
        addComment();
    } else if (e.getSource() == removeComment) {
        removeComment();
    } else if (e.getSource() == modifyComment) {
        modifyComment();
    } else if (e.getSource()
                == removeAllComments) {
        removeAllComments();
    }
}
```

```
// ProjectID: 9487, FunctionID: 8723069, ETS
public void actionPerformed(ActionEvent e) {
    File nextPage = null;
    for (Iterator iter = testFiles.iterator();
            iter.hasNext();) {
        File f = (File) iter.next();
        if (f.equals(currentDisplayed)) {
            if (iter.hasNext()) {
                nextPage = (File) iter.next();
                break;
            }
        }
    }
    if (nextPage == null) {
    }
}
```

Combining the experiment results and the answers to RQ2.1 and RQ2.2, we could conclude the answer to RQ2:

Answer to RQ2: The models exhibit poor generalization performance under cross-project observations. They may utilize project-specific features as shortcuts rather than learning generic code features to improve metric scores, which may be an underestimated challenge for code summarization. Evaluating the actual generalization performance of a model requires introducing a data-neutral test

set. However, the improvement in metric scores remains meaningful even if the test set contains projects that already appeared in the training set.

5 Discussion

It is commonly believed that code has some features that distinguish it from natural language, which are shared by code. Pre-trained models are pre-trained on datasets containing lots of codes and then fine-tuned on specific downstream tasks. Thus, in theory, the pre-trained models can fully learn the shared code features and have a strong generalization ability. In this work, however, the pre-trained model's cross-project generalization performance is surprisingly poor, which leads to our concern that the pre-trained models' ability to learn generic code features may not be satisfactory, while they may be relying on some project-specific information (e.g., programming style) as a shortcut to improve the metric scores.

There is no strict classification method for the various features contained in the code yet, nor are there metrics to directly evaluate the model's ability to learn these features. Therefore, we can only evaluate the model's ability to generalize cross-projects. Combined with work by Allamanis et al. [29], we take the naturalness perspective and link our observations to differences in the models' performance on different code channels: the models perform well on the explanatory channel but fail to catch up on the algorithmic channel. Ahead in the explanatory channel allows models to generate summaries familiar to developers, while ahead in the algorithmic channel allows models to generate summaries needed by developers.

In future work, we consider that the features contained in real code need to be categorized. From our observations, code may contain at least two types of features: generic code features and project-specific features. In addition, it is suggested that the model's performance under the different code channels should be further explored, and implicit code channels need to be explored.

6 Threats to Validity

The main threat to the validity of our work comes from generalizability. The pre-trained models chosen in this work are Transformer-based with parameters between 60M and 220M, which may not represent all pre-trained models in code summarization tasks. Our experiments are conducted on 19 projects from the Funcom [13] dataset. While we have tried to minimize the impact of particular projects, the model's performance on the 19 projects may not represent the situation on all projects. We use n-gram based metrics to align with existing studies, but the models' performance under the embedding-based metrics is also worth investigating.

7 Conclusion

Code comments are one of the pillars of software reuse, improving code readability and reducing maintenance costs. Code summarization aims to address the missing, outdated, and mismatched issues prevalent in comments by generating concise code comments. In this work, we evaluate the performance of models that cannot utilize the code's structured information by observing the cross-project generalization abilities of the latest pre-trained models. The results show that the models perform well on the explanatory channel but fail to catch up on the algorithmic channel. Further studies on the model's performance under different code channels and exploration of the implicit code channels are suggested. It is also anticipated that this work will be a reminder of the classification of code features and methods for evaluating the model's ability to learn code features.

References

1. Chen, Q., Xia, X., Hu, H., Lo, D., Li, S.: Why my code summarization model does not work: code comment improvement with category prediction. ACM Trans. Software Eng. Methodol. (TOSEM) **30**(2), 1–29 (2021)
2. Yau, S.S., Collofello, J.S.: Some stability measures for software maintenance. IEEE Trans. Software Eng. **6**, 545–552 (1980)
3. Hu, X., Li, G., Xia, X., Lo, D., Lu, S., Jin, Z.: Summarizing source code with transferred API knowledge (2018)
4. Xie, R., Hu, T., Ye, W., Zhang, S.: Low-resources project-specific code summarization. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1–12 (2022)
5. Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Summarizing source code using a neural attention model. In: 54th Annual Meeting of the Association for Computational Linguistics 2016, pp. 2073–2083. Association for Computational Linguistics (2016)
6. LeClair, A., Haque, S., Wu, L., McMillan, C.: Improved code summarization via a graph neural network. In: Proceedings of the 28th International Conference on Program Comprehension, pp. 184–195 (2020)
7. Vaswani, A., et al.: Attention is all you need. In: Advances in Neural Information Processing Systems, vol. 30 (2017)
8. Ahmad, W.U., Chakraborty, S., Ray, B., Chang, K.W.: A transformer-based approach for source code summarization. arXiv preprint arXiv:2005.00653 (2020)
9. Feng, Z., et al.: CodeBERT: a pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020)
10. Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., Yin, J.: Unixcoder: unified cross-modal pre-training for code representation. arXiv preprint arXiv:2203.03850 (2022)
11. Wang, Y., Le, H., Gotmare, A.D., Bui, N.D., Li, J., Hoi, S.C.: Codet5+: open code large language models for code understanding and generation. arXiv preprint arXiv:2305.07922 (2023)
12. Rahman, M., Palani, D., Rigby, P.C.: Natural software revisited. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 37–48. IEEE (2019)

13. LeClair, A., McMillan, C.: Recommendations for datasets for source code summarization. arXiv preprint arXiv:1904.02660 (2019)
14. Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P.: On the naturalness of software. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 837–847. IEEE (2012)
15. Zhang, C., et al.: A survey of automatic source code summarization. Symmetry **14**(3), 471 (2022)
16. Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q.V., Salakhutdinov, R.: Transformer-XL: attentive language models beyond a fixed-length context. arXiv preprint arXiv:1901.02860 (2019)
17. Zhou, Y., Shen, J., Zhang, X., Yang, W., Han, T., Chen, T.: Automatic source code summarization with graph attention networks. J. Syst. Softw. **188**, 111257 (2022)
18. Liang, Y., Zhu, K.: Automatic generation of text descriptive comments for code blocks. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 32 (2018)
19. Hu, X., Li, G., Xia, X., Lo, D., Jin, Z.: Deep code comment generation. In: Proceedings of the 26th Conference on Program Comprehension, pp. 200–210 (2018)
20. Wang, Y., Wang, W., Joty, S., Hoi, S.C.: Codet5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859 (2021)
21. Ahmad, W.U., Chakraborty, S., Ray, B., Chang, K.W.: Unified pre-training for program understanding and generation. arXiv preprint arXiv:2103.06333 (2021)
22. Haque, S., Eberhart, Z., Bansal, A., McMillan, C.: Semantic similarity metrics for evaluating source code summarization. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, pp. 36–47 (2022)
23. Papineni, K., Roukos, S., Ward, T., Zhu, W.J.: Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, pp. 311–318 (2002)
24. Lin, C.Y.: Rouge: a package for automatic evaluation of summaries. In: Text Summarization Branches Out, pp. 74–81 (2004)
25. Banerjee, S., Lavie, A.: Meteor: an automatic metric for MT evaluation with improved correlation with human judgments. In: Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization, pp. 65–72 (2005)
26. Croft, D., Coupland, S., Shell, J., Brown, S.: A fast and efficient semantic short text similarity metric. In: 2013 13th UK Workshop on Computational Intelligence (UKCI), pp. 221–227. IEEE (2013)
27. Reimers, N., Gurevych, I.: Sentence-BERT: sentence embeddings using SIAMESE BERT-networks. arXiv preprint arXiv:1908.10084 (2019)
28. Roy, D., Fakhoury, S., Arnaoudova, V.: Reassessing automatic evaluation metrics for code summarization tasks. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1105–1116 (2021)
29. Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C.: A survey of machine learning for big code and naturalness. ACM Comput. Surv. (CSUR) **51**(4), 1–37 (2018)

On Modularity of Neural Networks: Systematic Review and Open Challenges

Riku Alho, Mikko Raatikainen(✉), Lalli Myllyaho, and Jukka K. Nurminen

University of Helsinki, Helsinki, Finland
{riku.alho,mikko.raatikainen,jukka.k.nurminen}@helsinki.fi, lalli.myllyaho@alumni.helsinki.fi

Abstract. Modularity is used to manage the complexity of monolithic software systems and is a de facto practice in software engineering. Similar modularity concepts might also translate beneficially to machine learning, uncovering equivalent benefits, such as reuse opportunities. We address the research problem of modular neural networks' (MNNs) applicability, operations, and comparability to monolithic solutions. A systematic literature review is used to identify 86 studies that provide information regarding modularity compared to monolithic solutions. The selected studies address many tasks and domains, evidencing broad applicability, although applied modularity operations are limited mainly to splitting. Nearly two-thirds of studies show improvements in task accuracy compared to monolithic solutions. Only 16% of studies report performance values in their comparisons, but 82% report MNN performance benefits in training and inference time, memory, and energy consumption compared to monolithic solutions. However, publication bias can favor MNNs, and most studies were conducted in laboratory environments on focused tasks and static requirements. Nevertheless, we conclude that MNNs perform at least satisfactorily compared to monolithic solutions. Modularity can bring forth benefits for managing complexity and has the potential for development and operations performance efficiency. Therefore, modularity in neural networks opens research avenues for extending software reuse, especially regarding broadening the applicability of advanced solutions and experiences from long-term or industrial applications and use.

Keywords: modular neural networks · deep learning · modularity · systematic literature review

1 Introduction

In software engineering, modularity has been discussed since the early 1970s [19] and has become a de facto practice, e.g., to manage the complexity of large software systems. Modularity is also a central, although not alone sufficient, characteristic in several reuse approaches, such as in reusable components and software product lines. Modularity roughly means that a system or its part is

A. Achilleos et al. (Eds.): ICSR 2024, LNCS 14614, pp. 18–36, 2024.
https://doi.org/10.1007/978-3-031-66459-5_2

composed of separate, independent units or modules that can be easily connected, combined, or replaced. Modularity has appeared in different forms over the years and is still evolving in new approaches. One recent prominent example is the loosely coupled microservice architecture [12] that has recently been extended to data [16] and the user interface [17,20], covering basically the entire software system.

As modularity is essential to software, similar modularity concepts might also translate beneficially to *machine learning (ML)*, as ML could adopt a lot from software engineering [23]. This could also open new opportunities for adopting software reuse approaches and best practices, thus avoiding pitfalls. However, research on ML modularity, particularly on *neural networks (NN)* compared to monolithic solutions, is not comprehensively and systematically analyzed. To the best of our knowledge, only one relatively old survey outlines the general stages of *modular NN (MNN)* design, task decomposition techniques, learning schemes, and multi-module decision-making strategies [3]. No systematic reviews specifically focused on the MNN, where factors such as computing and maintenance costs could be alleviated through modularity. As a result, there is a need for a comprehensive synthesis of information regarding modularity in modern NNs. Such a synthesis would offer a comparative overview of the practicality and applicability of incorporating modularity into NNs.

This paper addresses the research problems of how MNNs are applied and whether MNNs can perform at least equally well as the monolithic NN solutions, bringing forward benefits of modularity, particularly reuse, that could be exploited. That is, if a similar inference is built as a monolithic and modular solution, such as a network of connected smaller NNs, how well do the solutions perform compared to each other? We synthesized the research evidence by conducting a systematic literature review [13] of studies that compare modularity with monolithic solutions. Through the review, we analyzed the domains and tasks to which the MNNs are applied, the applied modularity operations and their frequency, and how MNNs compare to monolithic neural networks in terms of performance measured by training and inference time, memory, and energy consumption.

Our analysis of the 86 included papers led us to conclude that MNNs perform at least satisfactorily compared to monolithic solutions without being limited to a specific application domain or task. Most studies focus on showing comparable accuracy, but some studies also show comparable performance in other metrics. However, almost all studies were conducted in academic settings, meaning that the results do not necessarily cover complex, evolving contexts, such as how coupled the modules of MNNs become over time, hindering the potential benefits. The operations also seem straightforward monolith splitting, leaving opportunities for more advanced operations and architectures. Nevertheless, the encouraging results call for future studies on possible practical benefits and novel application strategies of modularity.

The rest of this paper is structured as follows. Section 2 elaborates on the background, introducing modular operations as our lenses for analysis. In Sect. 3,

the systematic literature review as the applied research method is introduced. Section 4 gives an overview of the papers and presents and summarizes the results for each research question. The validity of the study is discussed in Sect. 5. Sections 6 and 7 contain the discussion and conclusions, respectively.

2 Background

Modularity or modular programming in software engineering emphasizes the aspect of separating the functionality of any large- or multi-functional system to consist of multiple independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality [19]. Often in industry, these modules are also given additional responsibilities beyond their core functionality. Industrial application demands, such as logging and security checks, often find their way into modules whose core responsibilities are something else. These industrial application demands are often called crosscutting aspect concerns because they tend to cut across multiple modules in a system. In addition to functional aspects, other standardized non-functional aspects may be translated as modularity attributes. These include performance, compatibility, capability, reliability, security, maintainability, flexibility, and safety [10]. Neither the terms module nor modularity are written as a separate technical taxonomy, e.g., in any of the software engineering standards, without the inclusion of being a part of a formal system breakdown structure [6]. This essentially means that the exact definitions for what module and modularity are for isolated entities outside of a functional relation with systems are still blurry, so we do not have standards to tell if something is a module without having it in the context "as a part of a system" beforehand. This also makes it difficult to analyze and categorize the modular quality of the inner workings of a module because the standards mainly point out the functionality appearance shown outside.

Modular Neural Networks (MNNs) are Neural Networks (NN). The exact definition of MNN is not absolute, but scientific literature includes multiple different structural designs [2], which seem to follow certain parts of modularity seen in software engineering. One generalizing explanation is that an MNN is a network that consists of multiple independent neural networks (modules) managed by some perhaps modular, crosscut aspect supporting intermediary program that inputs values to each network and takes their results [9] in some order or structural manner. Each module is dedicated to handling a subtask of the task the network as a whole tries to accomplish. On the other hand, not all modules need to be activated for all the tasks an MNN is given to handle, such as in hierarchical selection [9]. Some modules may also work independently in parallel, giving different outputs for a singular subtask; these can be structured in an ensemble majority vote, ensemble average vote, or hierarchically competitive designs [9]. Compared to regular neural networks, the main addition is the requirement for task decomposition and building networks for decomposed tasks, which can be done, e.g., pre-emptively and manually by the user or through some algorithmic means during training [11]. There is also scientific literature that separates

the notation of networks arranged in ensemble and modular combinations. Each individual network in an ensemble works to solve the entire problem. Different networks in the ensemble may use different methods and inputs. Results from multiple ensemble networks can be combined through averaging or other statistical methods. Multiple ensemble networks can provide better generalization and increased reliability through redundancy [18,24]. By contrast, modularity uses decomposition for problem-solving [8]. A solution should still be considered modular if ensembles are used in decomposed problem solutions.

We adopt and apply to the analysis of MNNs the work of Baldwin *et al.* [4], who identify and list six *modular design operations* in modular systems. Table 1 gives a summary listing and description of the modular design operations and elaborates on possible applications in the context of MNNs.

Table 1. Six modular design operations and their description [4] and application examples to MNNs.

Operation	Principle	MNN application example
Splitting	Modules can be made independent	A previously monolithic machine learning task can be split into subtasks and built as a cooperating group of independent networks. This is usually done through task decomposition, which can be done automatically through algorithms and methods, or manually
Substituting	Modules can be substituted and interchanged	With transfer learning, structurally improved student networks can serve as substitutes of teacher networks, or substitutional networks can be constructed through revised training data, simulation or oracles
Augmenting	New Modules can be added to create new solutions.	New networks can be added later to improve accuracy or for new tasks. Networks can be ensembled together to improve generalization
Inverting	The hierarchical dependencies between Modules can be rearranged.	Switching the order in which neural networks are run
Porting	Modules can be applied to different contexts.	Moving and using a neural network module in a different context. This usually is the case when the application domains are similar enough
Excluding	Existing Modules can be removed to build a usable solution.	Networks can be removed to build a usable solution

3 Research Approach

We apply the systematic literature review method [13] as the research approach in this paper. An essential step when performing a systematic literature review is the development of a protocol that specifies all steps performed during the review and increases rigor and reliability. The procedure starts with the definition of the research question, search strategy identification, and search scope selection. After that, study inclusion and exclusion criteria were formed based on the research questions. An empirical data extraction form is created based on the research questions. These procedures are elaborated below.

This study addresses the following research questions focusing on studies that compare monolithic and MNN solutions found in the scientific literature:

- **RQ1:** What domains and tasks have MNNs been used?
- **RQ2:** What modular design operations have been used with regard to neural networks?
- **RQ3:** How do MNNs compare to monolithic solutions?

The aim is to gain an overall understanding of the research field that compares MNN and monolithic NNs. RQ1 maps the different application domains and solution tasks of the MNN in order to assess whether solutions are applied or applicable to something specific or spread more generically over different domains and tasks. RQ2 characterizes the applied modularity solutions in terms of what modular design operations are used and their frequency by applying the above six modular design operations as the lenses of analysis. Finally, RQ3 reports how the MNNs perform against monolithic NNs and their metrics for each study.

We decided to conduct the search on two general-purpose databases, Scopus and Web of Science, and two databases focusing specifically on computer science, ACM digital library, and IEEExplorer. We straightforwardly constructed a search string from the research problem by including only the central words. We executed the following search string on metadata, i.e., title, indexing terms, and abstract, on the search engines applying their specific syntax:

> *("Modular neural network" OR "Modular neural networks") AND (Compare OR Comparing OR Comparison).*

Every selected paper had to meet the inclusion criteria (IC) below and none of the exclusion criteria (EC) below. In total, we found 484 papers and included 86.

- IC1: The paper experiments with MNNs. The experiments are required to collect information in order to analyze solutions, adoptions, and modular capability.
- IC2: The paper compares MNNs to Monolithic solutions. Comparisons are required for information on accuracy and efficiency.
- EC1: The paper does not feature the use of MNNs.
- EC2: The paper does not feature the use of Monolithic solutions.

– EC3: The paper is an editorial, technical report, position paper, abstract, keynote, opinion, tutorial summary, panel discussion, or book section.
– EC4: The paper is grey literature. Grey literature is argued to be of lower quality than papers published in journals and conferences as they usually are not thoroughly peer-reviewed [14].

Table 2. Data extraction form.

Data extracted	Topic
Bibliographic information	Overview
Evidence level	Overview
Monolithic Deep Learning	Overview
Application domain	RQ1
Application task	RQ1
Study type	RQ1
Modular design operators	RQ2
Monolithic accuracy	RQ3
MNN accuracy	RQ3
Computation time	RQ3
Memory	RQ3
Energy consumption	RQ3

Data was extracted from the selected studies using the data extraction form derived from the research questions in Table 2. We also link in Table 2 the reason for the extracted data and the location of full details either within the text section or in a table. Different data classifications, such as application domains and tasks, were formed from extracted data based on their emergence during analysis. However, for the evidence levels, the six-level classification system [1] was used, ranging from no evidence (score 1) to industrial application (score 6), addressing how simplified or realistic the research contexts are.

4 Results and Analysis

This section first gives an overview of the selected studies, and then the research questions are addressed by representing the extracted data and summarizing the data for each question.

4.1 Results Overview and Demographics

We included 86 papers[1] in the analysis after searches and applying inclusion and exclusion criteria. The bibliographic information of the selected studies is listed

[1] The full details are given in a separate appendix published in GitHub: https://github.com/ESE-UH/Systematic-literature-review-on-Modularity-in-Neural-Networks---Data-and-appendix.

in the supplementary material. Hereafter, we apply the prefix '*S*' to refer to the selected studies (i.e., S1–S86), and the studies are ordered by publication year. Figure 1 summarizes the number of papers per year. The first papers appeared in 1993, and the highest number of studies was published in 2009. Figure 1 indicates growth until 2010, and then it shows stagnation around 2011–2021. One reason can be those other machine learning solutions, such as monolithic deep neural networks, that gained popularity around 2011 through the increase in available hardware performance [15].

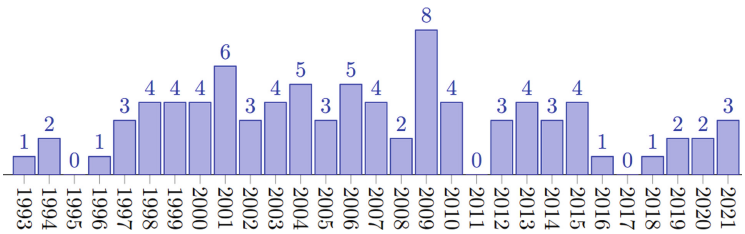

Fig. 1. The number of papers per year.

Deep learning is present as a specific technique of NN in 16 (18.06%) studies (S1, S65, S67, S68, S70, S72, S73, S74, S75, S76, S78, S80, S81, S82, S83, S84, S85). Since 2012, deep learning has been present in 33% of studies, increasing gradually and being present in almost all studies since 2019.

The evidence levels are as follows. Level 4 (Academic studies) is dominant, covering 82 papers (95.3%). The remaining papers S43 and S50 (2.3%) are at Level 3 (Expert opinions), and S1 and S66 (2.3%) are at Level 5 (Industrial studies).

4.2 RQ1: Modular Neural Network Solutions

Eighteen different application domains were identified from the papers (Table 3). The most popular application domain, Computer Science, consists of studies using general image databases, simulations, and games for their training, was in 34 (39.5%) studies. The rest of the domains are more specific but have significantly fewer studies.

Table 4 shows 19 identified tasks of the selected studies and the number of studies for each task. The number of studies is distributed to different tasks so that none of the tasks is clearly dominant.

Two study types – *Performance-type* and *Development-type* – were identified and differentiated. Performance-type studies try different ML solutions to improve the accuracy or efficiency of a certain task and compare them to other studies with solutions for that same task. Development-type studies pioneer a novel solution for a task and compare their results to others by developing and training their solutions for the task. Performance was discovered as the study

Table 3. The application domains in the selected studies.

Application domain	Number of papers(%)	Papers
Computer Science	34 (39.5%)	S6, S7, S12, S13, S14, S17, S23, S24, S29, S30, S32, S33, S35, S36, S39, S42, S44, S47, S48, S55, S59, S60, S62, S63, S71, S72, S73, S75, S76, S80, S82, S83, S84, S86
Medical	7 (8.1%)	S25, S26, S37, S61, S74, S81, S85
Chemistry	5 (5.8%)	S34, S43, S64, S65, S68
Economics	5	S1, S5, S41, S49, S52
Electrics	5	S3, S8, S22, S70, S78
Hydrology	5	S40, S45, S46, S51, S69
Physics	5	S4, S15, S20, S77, S79
Biology	4 (4.7%)	S2, S27, S66, S67
Transportation	4	S10, S16, S38, S56
Robotics	3 (3.5%)	S31, S50, S54
Meteorology	2 (2.3%)	S19, S58
Construction	1 (1.2%)	S11
Geography	1	S9
Geology	1	S28
Geophysics	1	S57
History	1	S53
Logic	1	S18
Surveillance	1	S21

type in 71 (82.6%) studies, while development type covered the remaining 15 (17.4%) studies (S18, S26, S37, S45, S50, S54, S62, S65, S66, S69, S70, S72, S76, S81, S83).

In summary, there is wide diversity in domains across studies. Computer science as a generic application domain is the most popular, and academic studies dominate the evidence levels, indicating a still ongoing development of MNN methods rather than maturing and industrial adoption. Researchers working in specific domains other than general computer science might be unaware or find it still infeasible or immature to apply these solutions to their domain. Wideness, however, suggests some interest in and potential of MNN solutions regardless of application domains. Likewise, the tasks provide even more extensive diversity across studies, and while time series was the most popular, image and input classification were almost as popular. This provides positive promise for the applicability of MNNs to a vast amount of tasks.

The identified development-type studies characterize the adoption of MNNs to new solutions where other viable options were considered. However, the lack of development-type primary studies is notable, as only 15 were found. High amounts of performance-type studies do not have the same impact in computer

Table 4. The tasks addressed by the selected studies.

Task	Number of papers (%)	Papers
Time series	11 (12.8%)	S1, S5, S8, S10, S15, S19, S20, S41, S46, S49, S52
Image classification	10 (11.6%)	S9, S13, S42, S53, S60, S61, S74, S81, S83, S84
Input classification	10	S25, S26, S29, S33, S35, S36, S37, S40, S48, S51
Pattern detection	9 (10.5%)	S2, S3, S7, S21, S22, S27, S28, S34, S78
Function generation/Input detection	8 (9.3%)	S11, S12, S16, S63, S64, S68, S79, S86
Multi-sensor classification	7 (8.1%)	S4, S23, S43, S44, S50, S54, S71
Model creation	6 (7.0%)	S14, S45, S56, S57, S58, S77
Input modeling/classification	5 (5.8%)	S65, S66, S67, S69, S70
Image detection	4 (4.7%)	S6, S30, S38, S47
Face detection	3 (3.5%)	S17, S39, S82
Sensor detection	3	S31, S72, S85
Image classification & detection	2 (2.3%)	S55, S80
Human recognition	2	S73, S75
Data classification	1 (1,2%)	S24
Image transformation	1	S76
Logic compression	1	S18
Pattern classification	1	S59
Text detection	1	S32
Video classification & detection	1	S62

science as the higher valued studies of development type, although their actual provided real-world value may be greater or the same as with development-type studies (cf. [5]). This makes it difficult for MNNs to gain notice. A higher amount of development-type comparison studies could provide MNNs with novel solutions and increase research and public interest.

4.3 RQ2: The Applied Modular Design Operations in the Research Literature

While all studies apply a modular design operation, most studies apply only splitting out of the six modular design operations. In Table 5, those studies that utilize design operations beyond just splitting are listed, along with descriptions of their modular design operations. While the oldest studies apply only splitting, no significant trend in the applied or number of modular design operations can be observed from an exploratory analysis over the years and chronologically increasing IDs.

As presented in Table 5, all operations have been applied successfully in an experiment, meaning that all six modular design operations can be applied in some tasks. However, splitting is the most prevalent, being covered by all except two MNNs in the selected studies. The other modular design operations were significantly rarer, being covered in 15 studies. S24, S31, S44, S50, S61, and S71 stand out as they covered at least four of the six modular design operations in

Table 5. The studies that utilize operations beyond just splitting, along with their modularity operations.

ID	Splitting	Substituting	Augmenting	Inverting	Porting	Excluding	Notes
S13	++	0	0	++	0	0	Monolithic task split to modules. Two type of cooperation among modular networks are considered: neural network and weighted combination of the modules outputs
S24,S61	++	++	++	++	0	++	Evolving and combining best performing modules from a generated neuron population
S31	0	++	++	0	0	++	Network evolves and expands through simulation events
S35	++	+	+	0	0	+	Monolithic task split to modules, mention of additional possible operations
S44	++	++	++	++	0	++	Evolving neural network modules, combining and removing them from agents in a simulated environment
S5	++	++	++	0	++	0	Simulated Robot locomotion on terrain, each leg own module, different legs
S54	++	++	++	0	0	0	Simulated Robot locomotion on terrain, each leg own module
S71	++	++	++	++	0	0	Evolving and choosing best performing neuron population modules
S72	++	0	++	0	0	0	Monolithic task split to modules, augment modules until satisfies some criteria
S80	0	+	0	++	0	0	Modules are formed through novel knowledge, evolves and expands through simulation events
S83,S84	++	+	+	+	+	+	Monolithic task split to modules, mention of other operators

(++) Applied in experiment. (+) Applied as discussion in theory. (0) Not applied.

their experiments. The small number of studies applying other operations might be caused by the way how most performance-type studies, i.e., the dominant type of studies, are currently structured. Performance-type studies focus on narrow tasks, resulting in MNNs being analyzed solely as splitting from a monolithic standpoint instead of exploring the additional concerns required by modular design operations. That is, most performance-type studies consist of tasks with static requirements that lack more dynamic requirements and complex contexts from industrial settings. There can also be a lack of publicly available ready-to-use transfer learning modules that could be used, e.g., in porting, augmenting, and substituting. Overall, no evidence was provided by the studies on any modular design operations being inapplicable to a task or domain, and no other modular design operation was discovered beyond the classification.

4.4 RQ3: Comparison of MNNs to Monolithic Solutions

The measures between MNN and monolithic NN were extracted from the studies. First, the accuracy of MNNs compared to monolithic solutions was analyzed. Second, the performance of modular neural network solutions in regard to execution time, and consumption of memory and energy were analyzed. The detailed results are given in the appendix (see footnote in Sect. 4.1), and summarized below.

All studies compare accuracy in one way or another. First, the accuracy percentage comparisons of MNNs to monolithic solutions are presented. Some studies iterated or compared MNNs in multiple configurations to multiple monolithic solutions, in which case the best-performing solution or configuration from both was considered, but we did not analyze these in detail. Second, the remaining accuracy measurements of studies provided alternate methods, such as error deviations, textual explanations, or compression size, instead of accuracy percentages to calculate their performance. 30 (34.9%) studies were found to use these alternate methods. Although solutions between studies cannot be directly compared in their accuracy to each other, their relative accuracy within each study can be counted and used as an effective measurement. We count that 57 (66.2%) studies have improved accuracy by using MNNs, 10 (11.6%) have seen a decline, and 9 (10.5%) have mixed or similar results.

We also extracted the information on the relative training time between MNNs and monolithic solutions, if available. 29 (33.7%) studies reported shorter training times for MNN solutions. 12 (14.0%) studies reported longer training times. Some studies reporting longer training times also mention the possibility of training NN modules in parallel with their task, which would shorten the overall training time. 45 (52.3%) studies have no information regarding training time.

The inference time-efficiency comparisons of MNNs to monolithic solutions are as follows. Only a small number of studies (14, i.e., 17.4%) include time-efficiency measurements. As with accuracy values, the time-efficiency values cannot be directly compared to other studies based on their own task and domain settings. Still, their relative results can be counted. Twelve studies report faster,

and two studies report slower average execution times for each run. Only four studies provide information regarding memory size and two quantify the size, and only two about energy consumption. All measures show smaller memory and energy consumption.

In summary, MNNs typically, although not always, perform better when compared to monolithic solutions in all analyzed measures, meaning that MNNs appear at least comparable to monolithic solutions. MNNs have better accuracy in nearly two-thirds of the studies. If only monolithic deep learning were assessed (for a list of studies applying deep learning, see Sect. 4.1), MNNs performed even better in 12 of the 16 countable studies, and two of the remaining four studies have mixed results. MNNs also provided positive promise on computing time, memory, and energy performance. However, these measures are significantly less common, which might be caused by the different historical public interests when the studies were conducted and the academic nature of the studies. Recent research advancements, industrial applications, and sustainability requirements have likely increased interest in different performance measures rather than only accuracy.

5 Research Approach Validity

In order to evaluate a systematic review's validity, Kitchenham *et al.* propose four quality questions for systematic reviews [1] that are discussed below:

"Are inclusion and exclusion criteria described and appropriate?" This review meets this criterion as it explicitly defines and explains its criteria.

"Is the literature search likely to have covered all relevant studies?" This criterion is partly met by searching four digital libraries. However, the most obvious threats are that our search did not apply other strategies, such as snowballing [25], and the string was a bit limited and did not cover all possible synonyms of the terms. Nevertheless, the resulting sample size of 86 studies is relatively significant. Another threat pertains to the fact that only studies comparing modular and monolithic solutions were included, which might leave out studies focusing solely on modularity.

"Did the reviewers assess the quality/validity of the included studies?" The quality and validity of studies were not extensively assessed due to the lack of data used to analyze them and since almost all studies were relatively similar academic studies. The applied evidence levels are only one indicator of quality and validity that pertains to how realistic the research settings are. We did not analyze or differentiate the various research methods or techniques applied.

"Were the basic data/studies adequately described?" This criterion is met as we used a detailed data collection form and reported full details for each study, partially in the online appendix.

In addition to the above four questions, a threat to the validity of our systematic review was the inaccuracy in data extraction. There were certain difficulties in extracting relevant information from selected studies. Several studies do not explicitly mention results for most of the fields covered in the extraction form,

such as computation time or memory. This could cause the researcher's interpretation bias to affect the final extracted data. The issue was mitigated by leaving highly interpretational results as *NA* (Not Available), which lowered the number of results and might affect the outcome.

Bias is, of course, another potential threat. Bias could affect the results, for example, by the fact that most of the paper selection and analysis was performed by the first author, under the supervision of the other authors. It is possible that some cases which the first author considered clear would not be clear to others. This applies to both paper selection and the analysis of the final set of papers. Some relevant papers may have been excluded based on how they were interpreted, and some analyses may not be entirely accurate because of how they were interpreted by the authors.

6 Discussion

6.1 Answers to Research Questions

The presented review shows the broad applicability of MNNs and the potential for further experiments and application. Even if there is no specific peak in research indicating very high research interest, the vast number of domains and tasks in RQ1 since 1994 suggests that MNNs have applicability in a wide range of problems. All six modular design operations are used at least in some context as the answer to RQ2. Still, the low number of performance studies and other operations than splitting, however, indicate quite straightforward modular design operations, just dividing a monolith into smaller pieces. This leaves open challenges for other kinds of operations.

While modular neural networks (MNNs) appear to perform well as the answer to RQ3, some caveats limit us from claiming that MNNs perform better than monoliths. It may be that researchers are more inclined to focus on MNNs and report results where the MNNs perform as well as or better than the monolithic models. The good performance may depend on the specific context in which they are applied, and they are not superior to monolithic solutions. The studies included in this analysis were interested in exploring the potential of MNNs and might have unintentionally identified better-suited solutions for MNNs than for monolithic models. It is also important to be mindful of publication bias, which may result in submitting and publishing results that favor positive results regarding MNNs. This bias could lead to the polarization of topics that have limited studies.

6.2 Application Opportunities and Limitations

Given their applicability and at least comparable accuracy and performance, MNNs appear as an interesting opportunity for attaining similar benefits to those of modularity in traditional software. In particular, reuse through modularity is a significant potential advantage for more efficient development and lower costs,

e.g., in training. Reuse can take various from intra-organizational reuse and customization to different customers to broader reuse over organizational borders through commercial and open source offerings. However, other advantages can also appear. For example, easier maintenance, retraining, and other aspects could also greatly benefit from modularity in managing the long-term iterative lifecycle of systems utilizing neural networks.

On the one hand, modularity can contribute to the characteristics of an ML system per se. In traditional software, separating concerns over different modules reduces the complexity of the system. Respectively in MNNs, having different modules responsible for their own tasks, the system becomes easier to understand, as a single function does not require crawling through the entire problem space. Smaller complexity of MNNs can also translate to improved transparency and a better understanding of how MNN modules operate and, thus, explainability of somewhat black box models. MNNs can also improve runtime performance by allowing for better parallelism utilization and smaller model sizes, which fit better into the processor's fastest but smallest memory, leading to faster and more cost-efficient computing.

On the other hand, modularity can significantly impact the development, operations, and maintenance processes as it enables the system to be improved, retrained, and assessed in smaller increments. By retraining one module at a time, possible benefits and failures can be identified earlier, without the need to wait for the entire model to be retrained, only to discover that the old model performed better. Moreover, by focusing on only a subset of the functionality, MNN can facilitate the evaluation of user value, including the level of accuracy or the degree of explainability that is desired.

Regarding modularity as a concept, operations beyond splitting, possibly beyond the applied taxonomy and not restricted to the modular concepts of software engineering, should also be considered. With this new context, researchers and practitioners should have the confidence to imagine and experiment with new ways to modularize their systems and not limit themselves to old approaches. In particular, modularity is not just about splitting a monolith into pieces in a pipe-and-filter architectural style, where one module's output is the next module's input. Instead, a more advanced architecture for MNNs can be constructed, such as a directed acyclic graph (DAG), where different modules consume output as input depending on their needs, or modules can be organized into ensembles to conduct advanced operations, such as voting.

As MNNs seem comparative to monolithic solutions, possible disadvantages of MNNs pertain to their continuous development and operations. This is often called MLOps, a DevOps derivative covering iterative and incremental development, retraining, integration, and deployment practices. While modularity can facilitate incremental development, a key measure of modularity is how tightly the modules are coupled to each other. Tight coupling can result in changes to one module, inducing changes to other modules, thus hampering the benefits of modularity. Modularity should not only mean splitting a monolith but also designing loosely coupled MNNs where different modules can be retrained

at their own pace, regardless of other modules. In particular, testing and monitoring can be more straightforward and transparent for smaller modules, but combining the results may pose challenges. Therefore, it is essential to investigate how different modules can be combined effectively and efficiently.

MNNs demonstrate versatile applicability even in complex use cases, holding potential in scenarios demanding real-time processing and resource efficiency, such as within aircraft or spacecraft flight management systems [7]. In these contexts, they offer benefits like enhanced situational awareness, automation, predictive capabilities, and decision support through collaborative decision-making processes. Moreover, their efficacy in robotic sensing (S31, S50, S54) underscores their adaptability across various domains. Expanding research could explore their integration into scalable drone swarms or similar modular units, fostering cooperative decision-making among distributed entities. Notably, drones could facilitate the deployment and iteration of MNNs, enabling the adoption of newer models or ensemble strategies within swarm frameworks.

6.3 Challenges of Industrial Applications

With the academic studies dominating our sample, more practical research is needed. Not only are the aforementioned potential benefits primarily practical, but the datasets used in the academic studies also may not reflect the industrial reality. The lack of industrial research also makes it challenging to recognize the effects of MNNs as a long-term solution that needs maintenance and evolves to serve different needs. Further industrial research would better assess whether the MNNs actually match the capabilities of monolithic solutions. The comparisons in industrial settings would not only show how accurate MNNs are in practical settings but could also give a better understanding of other potential benefits and shortcomings through other measures.

Also, due to limitations in research contexts, we didn't directly find any implications in our sample assessing other industrial-level concerns regarding modularity. As a general note, when applying modularity in MMNs, any horizontally affecting functional concerns, such as logging, which generally crosscut multiple modules, may be applied to MNNs. However, for entanglement and performance reasons, they should be kept purely on the intermediary layer, or split as additional crosscut plugin modules to each individual MNN. As for non-functional concerns in the light of SQuaRE taxonomy [10], generally illustrated benefits of modularity in software engineering should apply mostly painlessly to MNNs, especially in a scenario where the intermediary layer supports decentralized computing:

- **Performance:** MNN modules can be run individually, enabling workloads with less memory and computation power required.
- **Compatibility:** Smaller, non-entangled MNN modules are simpler to comprehend and made compatible with each other.
- **Reliability:** Multiple separate machines can replicate the same MNN module, in case of hardware breakdown.

- **Security:** Intermediary layer can validate and ensure the security of the whole MNN system and its flow.
- **Maintainability:** Individual MNN modules are simpler to understand, test, modify, maintain, and replace in production.
- **Scalability:** MNN modules can be decentralized to multiple separate machines, and multiple separate machines can replicate the same NN module to share the flow workload. Modules can be transferred individually from one environment to the other.
- **Safety:** Intermediary layer can validate and ensure the safety of the whole MNN system and its flow.
- **Compliance:** Requirements are simpler to fill and verify for smaller individual MNN modules.

The reasons for the lack of industrial evidence are difficult to pinpoint. The decline in the annual number of studies after 2009 could suggest that the rise of deep learning – the development of which has been heavily driven by enormous monolithic models – has driven attention away from MNNs, as the decline coincides with the rise of deep learning. In 2009, machine learning was made more readily available through the increase in computation availability with the introduction of the manufacturer (Nvidia) support for graphics processing unit (GPU) use in general-purpose calculations [21]. Since then, ImageNet has stirred public interest in deep learning through ImageNet Large Scale Visual Recognition Challenge (ILSVRC) competitions [22], where deep learning solutions provided the best leaps in the accuracy of results. There is also currently a lack of available tutorials, documentation, and guides for MNNs generally and in popular machine learning libraries and platforms, such as TensorFlow, Keras, and PyTorch. This may limit the appearance of such solutions in the industry, especially in smaller companies. One option would be to invest in the creation of an entirely new set of tools that focuses more on supporting modularity in machine learning.

Of course, a lack of industrial evidence in research literature does not necessarily mean a lack of evidence in the industry. There might not be industrial research publications as the research might be tightly shut behind corporate walls, or some companies may even use MNNs actively, but the information has yet to reach the academic community due to a lack of empirical research. It is also possible that the information has not been published because the results have not been encouraging, as people are often more inclined to publish success stories than failures.

7 Conclusions and Future Work

In this literature review, we have shown that modularity in MNN has been a research topic for over three decades, providing solutions comparable to monolithic solutions, which still appear as more dominant solutions in research and practice. However, AI is becoming more commonly used in industrial practice,

applied to various domains, and increasingly demanding tasks. This results in an incremental increase in the complexity of ML models, the size of ML-based systems, and the need for continuous and iterative practices in MLOps. Modularity can bring forth benefits for managing complexity and be a key enabler for various reuse strategies both intra-organizationally and by commercial and open-source offerings. Even more broadly, AI models are getting more complex, large, and resource-intensive, such as in the recently popularized language models and generative AI, which could benefit from modular and reuse design practices. Modularity has the potential for development and operations (MLOps) and maintenance efficiency.

As for future work, we point out several directions. Modularity deserves more attention regarding more advanced solutions and experiences from industrial application and use. That is, most studies are based on academic research, which can simplify the challenges and solutions. Academic research typically lacks long-term considerations of maintenance and iterative development, which might be a major hindrance to modularity compared to monoliths in the case of ML. Further studies and efforts should be focused on the identification of a good general method for finding MNN solutions to tasks suitable for NNs. This could, for example, be studied in the utilization of ever-developing large foundational models that require huge computational costs to retrain or fine-tune as a whole; using the foundational model only as a reused base model for smaller, more focused models could leverage the foundational model's enormous capabilities in information extraction in minute, yet intricate tasks that the foundational model itself is not trained to handle as is. Work should also be done on developing scalable intermediary programs capable of ad hoc distributed MNN machine learning; current open-source libraries have been written to rely only on monolithic computational architectures. More well-rounded standards for evaluating machine learning solutions should be created and adopted, which would value other aspects in addition to accuracy, for which sustainability is one direction. Research on more power-conservative solutions, such as MNNs, combined with less complex deep learning modules can be useful to maintain global energy sufficiency.

Acknowledgements. This work was partially funded by Business Finland in the IML4E project of ITEA4/EUREKA Cluster.

References

1. Alves, V., Niu, N., Alves, C., Valença, G.: Requirements engineering for software product lines: a systematic literature review. Inf. Softw. Technol. **52**(8), 806–820 (2010)
2. Auda, G., Kamel, M.: Modular neural network classifiers: a comparative study. J. Intell. Rob. Syst. **21**, 117–129 (1998)
3. Auda, G., Kamel, M.: Modular neural networks: a survey. Int. J. Neural Syst. **9**(02), 129–151 (1999)
4. Baldwin, C.Y., Clark, K.B.: Modularity in the design of complex engineering systems. In: Complex Engineered Systems, pp. 175–205 (2006)

5. Boulesteix, A.L., Lauer, S., Eugster, M.J.: A plea for neutral comparison studies in computational sciences. PLoS ONE **8**(4), e61562 (2013)
6. Efatmaneshnik, M., Shoval, S., Qiao, L.: A standard description of the terms module and modularity for systems engineering. IEEE Trans. Eng. Manage. **67**(2), 365–375 (2018)
7. Emami, S.A., Castaldi, P., Banazadeh, A.: Neural network-based flight control systems: Present and future. Annu. Rev. Control. **53**, 97–137 (2022). https://doi.org/10.1016/j.arcontrol.2022.04.006
8. Happel, B.L., Murre, J.M.: Design and evolution of modular neural network architectures. Neural Netw. **7**(6–7), 985–1004 (1994)
9. Hrycej, T.: Modular learning in neural networks: a modularized approach to neural network classification (1992)
10. ISO/IEC: Systems and software engineering – Systems and software quality requirements and evaluation (SQuaRE) – System and software quality models. ISO/IEC 25010, International Organization for Standardization, Geneva, Switzerland (2011)
11. Jacobs, R.A., Jordan, M.I., Barto, A.G.: Task decomposition through competition in a modular connectionist architecture: the what and where vision tasks. Cogn. Sci. **15**(2), 219–250 (1991)
12. Jamshidi, P., Pahl, C., Mendonça, N.C., Lewis, J., Tilkov, S.: Microservices: the journey so far and challenges ahead. IEEE Softw. **35**(3), 24–35 (2018)
13. Kitchenham, B., Charters, S.: Guidelines for performing systematic literature reviews in software engineering (2007)
14. Kitchenham, B.A., et al.: Refining the systematic literature review process-two participant-observer case studies. Empir. Softw. Eng. **15**(6), 618–653 (2010)
15. LeCun, Y.: 1.1 deep learning hardware: past, present, and future. In: 2019 IEEE International Solid-State Circuits Conference-(ISSCC), pp. 12–19. IEEE (2019)
16. Loukiala, A., Joutsenlahti, J.-P., Raatikainen, M., Mikkonen, T., Lehtonen, T.: Migrating from a centralized data warehouse to a decentralized data platform architecture. In: Ardito, L., Jedlitschka, A., Morisio, M., Torchiano, M. (eds.) PROFES 2021. LNCS, vol. 13126, pp. 36–48. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-91452-3_3
17. Männistö, J., Tuovinen, A.P., Raatikainen, M.: Experiences on a frameworkless micro-frontend architecture in a small organization. In: IEEE International Conference on Software Architecture. Software Architecture in Practice (SAIP) track (2023, accepted/in press)
18. Myllyaho, L., Raatikainen, M., Männistö, T., Nurminen, J.K., Mikkonen, T.: On misbehaviour and fault tolerance in machine learning systems. J. Syst. Softw. **183**, 111096 (2022)
19. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. In: Pioneers and Their Contributions to Software Engineering, pp. 479–498 (1972)
20. Peltonen, S., Mezzalira, L., Taibi, D.: Motivations, benefits, and issues for adopting Micro-frontends: a multivocal literature review. Inf. Softw. Technol. **136**, 106571 (2021)
21. Raina, R., Madhavan, A., Ng, A.Y.: Large-scale deep unsupervised learning using graphics processors. In: International Conference on Machine Learning, pp. 873–880 (2009)
22. Russakovsky, O., et al.: ImageNet large scale visual recognition challenge. Int. J. Comput. Vision **115**(3), 211–252 (2015)

23. Serban, A., van der Blom, K., Hoos, H., Visser, J.: Adoption and effects of software engineering best practices in machine learning. In: International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–12 (2020)
24. Sharkey, A.J.: Combining Artificial Neural Nets: Ensemble and Modular Multi-net Systems. Springer, Cham (2012). https://doi.org/10.1007/978-1-4471-0793-4
25. Wohlin, C.: Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Evaluation and Assessment in Software Engineering, pp. 1–10 (2014)

Generative AI for Code Generation: Software Reuse Implications

Georgia M. Kapitsaki(✉)

Department of Computer Science, University of Cyprus, Nicosia, Cyprus
gkapi@ucy.ac.cy

Abstract. Generative AI has lately started being used in the software engineering process. Developers are relying on ChatGPT, GitHub Copilot or other tools to accelerate the development process. Previous works have provided an overview of the tools and have compared their capabilities. Nevertheless, the relation with software reuse in the framework of Generative AI has not been examined extensively. In this work, we are studying how generative AI techniques respond to and affect software reuse, with an emphasis on software licensing issues and end-users' data privacy. We are using the following five tools: OpenAI ChatGPT, Google Gemini, GitHub Copilot, TabNine and Amazon CodeWhisperer. We provide an overview of the tools and previous works that have used them when examining code generation, discuss the implications on software reuse, and use a simple front-end use case to showcase how they respond on licensing and end-users' data privacy issues. This work introduces also a conceptual model that can help in improvements in the discussed reuse aspects.

Keywords: Generative AI · software reuse · ChatGPT · GitHub Copilot · Google Gemini

1 Introduction

Generative AI (GenAI) has currently started being used also in the software engineering process [21]. It is a disruptive technology that is expected to radically change the way software development is performed, and has already changed the usual way of performing things. Developers have now the opportunity to rely on ChatGPT or other software engineering dedicated tools like GitHub Copilot to accelerate the development process [5]. Such tools can be beneficial in assisting in repetitive tasks (generating test cases, validating requirements) but also in speeding up the main source code implementation and fixing software bugs [7].

Previous works have provided an overview of the tools for software practitioners and their capabilities and have compared their capabilities in terms of quality of generated code [3,4,13,23]. Other works have used more specific examples to investigate the properties of the generated code, such as in the area of software product lines [1]. Nevertheless, the relation between Generative AI tools and

A. Achilleos et al. (Eds.): ICSR 2024, LNCS 14614, pp. 37–47, 2024.
https://doi.org/10.1007/978-3-031-66459-5_3

software reuse has not been examined extensively. The associated intellectual processing rights (IPR) issues in that context is one main risk of these facilitators. For instance, open source software (OSS) can be reused but under specific conditions, as it needs to follow the conditions of the accompanying software license. Generative AI tools have been accused for lack of 'fair use' for being trained on code the creator of the software does not own [6].

In this work, we are studying how generative AI tools for code generation are related with software reuse with an emphasis on software licensing issues and end-users' data privacy. We provide an overview of the tools, and discuss the implications on software reuse introducing also a conceptual model with useful mechanisms for responsible software reuse. In order to showcase the usage of the tools we also list the literature where they have been used for code generation and present their results when used in a simple front-end application use case. We are focusing on the following five tools: OpenAI ChatGPT, Google Gemini, GitHub Copilot, TabNine and Amazon CodeWhisperer. The tools used been selected based on their popularity and on whether a free (or at least a free trial) version of them is available so that they can be used for comparison purposes within this work. We argue that this initial work can serve in better understanding the software reuse implications in the context of the studied Generative AI tools and can trigger further activities towards reuse-aware tools and responsible usage.

2 Software Reuse Implications and GenAI Tools Overview

When it comes to software reuse, there are two areas that are very important, as they affect how software is used and whether it causes major issues to its users, so we examine GenAI tools for code generation in respect to these areas, and list specific implications that need to be clarified in each area:

- **Open Source Software licensing**: OSS carries a specific license that defines the terms under which the original software can be used (by the licensees), although multi-licensing schemes also exist [12,15]. Various open source licenses exist, ranging from permissive to weak- and strong-copyleft, with some of them being more popular and most widely used (MIT is the more popular permissive license, while GNU GPL-2.0 and GNU GPL-3.0 the most popular strong-copyleft licenses). Software that carries specific licenses needs to be reused correctly, as failure to comply to the software's license restrictions may have legal consequences. Relevant cases of court disputes can be found in the literature, such as the Jacobsen versus Katzer case [19]. Organizations are devoting part of their resources on license compliance, while there are available open source and proprietary tools that assist in automating this process (e.g. Artifactory, FOSSology [10], findOSSLicense [11]).
  - **License of source code the tool trained on.** Does the tool consider the licensing of the source code it uses? This is a major issue in software reuse, and a previous work on a research agenda in Generative AI for

software engineering introduced the following research question that is related with this and the next implication: "*How does the integration of GenAI in software-intensive businesses affect software licensing models, and intellectual property rights?*" [17]. For instance, GitHub Copilot has been questioned on whether there was 'fair use' of the code it trained on, when it comes to licensing.[1]

- **License that can be used on the generated source code.** Does the generated code need to be made open source? Most probably yes, as tools have trained on public code data, as has also been argued.[2] Another side issue is whether the tool considers other IPR issues, such as patents and trademarks, although we do not examine it further in the context of the use case..

Security and privacy warranties.

– **End-users' data privacy**: Software needs to respect end-users' data privacy, and apply also relevant security mechanisms to guarantee this. Privacy laws, such as the EU General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA), have had a major effect on software creation, as software needs to comply with these laws, informing users on how their data are being collected, stored, processed and used (referred to as the *right to information* in GDPR) and offering at the same time ways to exercise user rights (e.g. *right to be forgotten*, *right to opt-out*) [14].

- **Security and privacy warranties.** Some tools indicate that they respect user's privacy so that the generated code is not disclosed (e.g. in TabNine). However, this refers to the usage data coming from developers. Which privacy aspects the tool considers when it comes to the used and generated code is less clear. Another important implication is whether the tool considers relevant privacy laws. When it comes to security, an important question is whether the tool provides mechanisms that verify that the generated code can be trusted, i.e. ensuring that it does not include any security loopholes that were intentionally or unintentionally introduced. Overall, GenAI tools are believed to have been trained on source code that may contain vulnerabilities [17].

An overview of the tools reuse features is presented in Table 1, whereas the use of its tool for code generation in the literature is presented below.

OpenAI ChatGPT. ChatGPT[3] (Chat Generative Pre-trained Transformer) is the GenAI tool most widely used. It relies on GPT-4 that shares the main principles as its previous version (its current free version uses GPT-3.5). It is based on a transformer-based neural network that handles and generates natural language text [20]. Its purpose is not software engineering, but it can be used to generate source code snippets. It is not intended to consider OSS licensing, or produce secure and privacy-aware code. Concerning user's data, it does store account

[1] https://sfconservancy.org/blog/2022/feb/03/github-copilot-copyleft-gpl/.
[2] https://goldin.io/blog/generated-code-fair-use.
[3] https://chat.openai.com/.

information and the chat conversation history. ChatGPT has been tested on different scenarios of use. It has been used in architecture-centric software engineering for a microservices-based software [2]. It has also been used in a use case concerning how LLM-based assistants can support domain analysts and developers, where different prompts were used until the results were satisfactory [1]. Overall, ChatGPT is the tool most studied in the literature in relation with software engineering. It has been employed for bug-fixing where it was found that it is competitive to the CoCoNut and Codex deep learning approaches [22], and for code correctness in EvalPlus that combines LLM- and mutation-based input generation for evaluating the correctness of generated code [13].

Google Gemini. Gemini[4] uses LaMDA (Language Model for Dialogue Applications) that employs a decoder-only transformer language model. It is pre-trained on 1.56-T words coming from data of public dialog and web text. Via the export to Replit feature, it supports 18 programming languages. Data are collected by default (conversations, location, feedback, usage information) but this can be turned-off (in that case conversations are saved for up to 72 h). Concerning code generation, there is an indication to use with caution and for licensing and code repositories, the following is indicated: "*In the case of citations to code repositories, the citation may also reference an applicable open source license.*" Since it is more recent than ChatGPT, it has not been examined much in the existing literature concerning its usage for code generation. It was compared against GPT-3.5 on Java code correctness, using a number of code assignments from codingbat.com, and it was found that GPT-3.5 performed better [4].

GitHub Copilot. GitHub Copilot[5] uses the GPT-3.5 Turbo model to offer coding suggestions. Copilot has been trained and fine-tuned mainly using public GitHub repositories and publicly available source code. In order to assist with licensing, Copilot can filter out code that has a match with a specific source of 150 characters or more. Concerning data privacy, prompts and code suggestions are saved by default, but the user has the chance to indicate preferences concerning collection, retention, and processing of data. There are no specific mechanisms for secure and privacy-aware code. Copilot was used to generate code in a number of scenarios with cybersecurity weaknesses using the MITRE's top 25 Common Weakness Enumeration (CWE) list and vulnerabilities in 40.73% of the total provided options were found [18]. Productivity and code quality when using GitHub Copilot has been investigated via a pair programming experiment and it was found that Copilot increases productivity (measured by the metric of lines of code added), but the quality of the code produced is not of the same level, as more lines of code need to be deleted (in the conducted empirical experiment) [9]. In order to examine the correctness of suggested code, 33 LeetCode questions were used to create queries using four programming languages [16]. SonarQube's cyclomatic complexity and cognitive complexity metrics were used to measure

[4] https://gemini.google.com/app.
[5] https://github.com/features/copilot.

code understandability and it was found that Java suggestions had the highest score (57%), while JavaScript the lowest (27%).

TabNine. TabNine[6] is trained on open source software licensed under a permissive OSS license (MIT, MIT-0, Apache-2.0, BSD-2-Clause, BSD-3-Clause, Unlicense, CC0-1.0, CC-BY-3.0, CC-BY-4.0, RSA-MD, 0BSD, WTFPL, ISC). Nevertheless, information on its model are not disclosed to a wide extent. Concerning privacy, TabNine indicates that no source code of the developer is retained or shared with third parties. It gives emphasis on security, as it runs on secure containers in Google Kubernetes Engine. For the case of Java, 100 methods obtained from open-source projects were used in order to compare TabNine, GitHub Copilot, ChatGPT and Bard [3]. The quality of the generated code was examined using functional correctness, complexity, efficiency, and size. There was no clear winner among the tools.

Amazon CodeWhisperer. Amazon CodeWhisperer[7] is trained on data including Amazon's and open source code. It can identify suggested code that is similar to OSS repositories (provides its URL), so that the user can check license issues, and consider the license of the software (adding e.g. appropriate attribution). It can also scan the code and indicate security issues by using static application security testing, secrets detection, and infrastructure as code (IaC) scanning. A previous work compared GitHub Copilot, Amazon CodeWhisperer, and ChatGPT in terms of code quality metrics (e.g. code validity, code correctness, code security, code reliability) using the HumanEval dataset that contains 164 problems [23]. It was found that the tools provide correct code 65.2%, 46.3%, and 31.1% of the time respectively. Amazon CodeWhisperer was the only tool that would present the source of the recommendation, whereas all generators succeeded in generating secure code. Another similar work focused on the properties of correctness, efficiency and maintainability of generated code versus human code in Java, Python and C++ using problems from LeetCode as in other works [8]. A variety of metrics were used including time and space complexity, memory usage, Lines Of Code and cyclomatic complexity. ChatGPT, BingAI Chat, GitHub Copilot, StarCoder, Code Llama, CodeWhisperer and InstructCodeT5+ 16b were used. It was found that the generated code solved the respective problem in 20.6% of cases: GitHub Copilot performed better than the other tools and CodeWhisperer did not manage to solve any problem.

3 Use Case Comparison

We are comparing the available tools for prompt engineering using their chat capabilities, via a simple use case of a feature typical in a CRM (Customer Relationship Management) system: helpdesk customer service automation. Specifically, we want to create a form that allows customers to input their helpdesk

[6] https://www.TabNine.com/.
[7] https://aws.amazon.com/codewhisperer/.

Table 1. Overview of tools and software reuse features.

	ChatGPT	**Gemini**	**GitHub Copilot**	**TabNine**	**CodeWhisperer**
Launch	Nov. 30, 2022	Mar. 21, 2023	Oct. 2021	2018	Apr. 13, 2023
Supported programming languages	No official list	Bash, C, C#, C++, CSS, Dart, Go, HTML, Java, JavaScript, Kotlin, PHP, Python, Ruby, Rust, SQL, Swift, TypeScript	best for C#, C++,Go, JavaScript, Python, Ruby, TypeScript	ABAP, C, C#, C++, CSS, Cuda, Go, Groovy, Java, JavaScript, Julia, HTML5, Kotlin/Dart, Lua, Matlab, PHP, Perl, Python, React/Vue, R, Ruby Rust, SQL, Scala, Shell, Swift, Terraform, TypeScript, VB	C, C#, C++, Go, Java, JavaScript, Kotlin, PHP, Python, Ruby, Rust, SQL, Scala, Shell, TypeScript
OSS Licensing	-	When citing repositories, license may be indicated	Can filter out code with match from a source of more than 150 characters, will add support for licensing filtering	Trained on OSS code with permissive licenses	Indicates used OSS repositories URL so that the developer can check the licenses
Developer's privacy	Stores conversation history, can be turned off	Stores prompts, location etc. by default, can be turned off	Stores prompts, suggestions, user can indicate preferences	Does not store developer's code or context window	Follows AWS model
Code security	-	-	May produce vulnerable code [18]	Runs on secure containers (Google Kubernetes Engine)	Can perform static application security testing, secrets detection and IaC scanning

request. We are also interesting in licensing compliance and end-users' data privacy protection (we are not exploring code security further). We develop the above use case in a number of user consecutive prompts:

1. Give me the front-end code in JavaScript that allows a user to add a helpdesk request, asking for a specific issue to be resolved, and also allows the user to view his requests and status.
2. The code needs to comply with GDPR. Please add implementation for relevant user rights, such as the right to be forgotten.
3. Does the above code comply to privacy legislation? EU GDPR? CCPA?
4. What about right of access? Can you suggest code for this user right?
5. Which is the software license of the code you gave me?

6. Can I reuse this code? Which open source software license should I use? I would like to use GPL-3.0 license if feasible.
7. Can I use the MIT license instead?

Concerning end-user's data protection, the relevant prompts refer to compliance with the legislation and use two user rights: right to be forgotten (or right to erasure) found in GDPR and CCPA, and right of access where the end-user can request access to his/her data found in GDPR. For the license prompts (#6 to #7), we included one permissive license (MIT) and one strong-copyleft license (GPL-3.0). We used Visual Studio Code for the GitHub Copilot, TabNine and Amazon CodeWhisperer plugins. We are listing below how each tool handles the request for the right to be forgotten (but do not present the respective code):

- ChatGPT, GitHub Copilot, TabNine, CodeWhisperer: delete option available next to each helpdesk request.
- Gemini: adds function that finds the request by ID and replaces the issue description with a message indicating anonymization, while updating the request status.

For prompt #1 ChatGPT provided a separate CSS file, while Gemini, Copilot and TabNine gave more detailed code explanations. GitHub Copilot also gave suggestions for what to ask in order to improve the helpdesk, such as implementing authentication and authorization for the helpdesk requests. For Amazon CodeWhisperer (using Q chat), the first user prompt had to be divided into three prompts, as CodeWhisperer was unable to provide a solution. That was feasible with the separate prompts. For prompt #2 it was possible to get a code suggestion with CodeWhisperer after using different prompts and the following worked (while other attempts did not): *Please give me the code for the GDPR right to be forgotten for the helpdesk code.* When asked to generate GDPR-compliant code, only the provided example of the right to be forgotten was added by the tools, although the prompt provides the right to be forgotten only as an example. Gemini suggested code that deletes the request status of a specific request, instead of deleting it completely. None of the tools provided the option to delete all helpdesk requests at the same time.

When asking about GDPR and CCPA compliance (prompt #3), CodeWhisperer was unable to answer. Gemini also recommended ways to address GDPR and CCPA compliance (e.g. *privacy policy*, *appropriate security controls*,*data breach response plan*). ChatGPT, Copilot and TabNine indicated other requirements for GDPR and CCPA compliance, such as consent, data minimization, privacy policy, and access and portability, but would not suggest specific ways to implement them, as Gemini did. For prompt #4, ChatGPT and Gemini provided an alert box with the requests, while Copilot offered to download them as a JSON file that would be a more suitable solution. TabNine mentioned that it added a new function (*getHelpdeskRequestDetails*) but no such function was present in the respective code. CodeWhisperer only provided a function that retrieves user requests. Overall, tools' replies are based on the data they are trained on but do not have understanding of legal issues or compliance.

Concerning licensing of the suggested code (user prompt #5), ChatGPT indicated that the code "*is not associated with any specific software license*" and provided some license examples. Gemini did not indicate the license of the source code used, as it said that it is "*unable to assign a specific software license to the code.*" As the use case and the generated code snippets are small, no specific repository was cited, as in that case the license would also be indicated (for Gemini). Neither Copilot was very clear about the source code used: "*The code I provide is generated on-the-fly and doesn't come from a licensed software product*", while TabNine and CodeWhisperer said that the license is not specified. So overall, the tools were not clear about the license of the suggested code.

When asked for the use of a specific license in user prompts #6 to #7, in ChatGPT, GitHub Copilot and TabNine the use of any of the licenses indicated (GPL-3.0, MIT) was considered feasible. Gemini mentioned that GPL-3.0 can be applied if a fully functional helpdesk system is created but it might not be so suitable for the code snippets. MIT is suitable according to Gemini. Only TabNine indicated that in order to use any of these licenses, the code needs to be adapted to "*include a copyright notice and a license header.*" Concerning the user prompt #6 on licensing, the answer provided by Amazon CodeWhisperer was generic, providing information on the GPL-3.0 license. It mentioned however, that MIT can be used on any suggested code snippets.

4 Conceptual Model for Software Reuse Implications

Considering the implications listed earlier and after studying the available tools, we have devised a conceptual reuse responsible model for Generative AI in software engineering. The model consists of mechanisms depicted in Fig. 1:

- **Train data indication and options**: the tool should inform users about the data it was trained on, in terms of whether it was trained on open source or commercial software (*Source indication* mechanism) and could even provide the user with some options, e.g. using data with only permissive licenses (*Source selection* and *Source licensing options* mechanisms). This requires nevertheless a lot of resources and time due to the intensive training phase that cannot be repeated, so instead the provision of a limited number of options might be more feasible (using a limited number of alternative pre-trained models). GitHub Copilot has announced that in this direction it will allow to see an inventory of similar code found in GitHub public repositories and the user will be able to sort the inventory by license (among other options).[8]
- **Secure train data**: the use of external tools can be used to restrict the source code data used for training using sources for well known software vulnerabilities, such as MITRE's Common Vulnerabilities and Exposures that have been used in previous works [18] to compare the tools, and VirusTotal

[8] https://github.blog/2022-11-01-preview-referencing-public-code-in-github-copilot/.

so that use of vulnerable code is limited. SonarQube's Security Module was also used to examine generated code's security, so its integration might be useful [23].

- **Legal framework**: since specific countries fall under different law jurisdictions that might affect the software compliance (e.g. providers operating in EU need to comply to GDPR and offer the relevant user rights, whereas in some US areas CCPA needs to be considered), relevant laws when it comes to data privacy need to be considered so that the generated code is also privacy-aware in a legal way (*Privacy legal framework* mechanism). The same can be applied in other laws, e.g. for copyright protection (*Other local legal restrictions* mechanism).
- **Integrated information**: the tool can provide the user with some options on the licensing scheme that can be chosen (*License recommendation* mechanism). In the same direction, adding to the generated code by default attribution to the source that assisted in generating the code might be useful (*Source attribution* mechanism). Source code quality metrics and relevant tools that measure them (e.g. SonarQube) can be integrated to the generative AI tools so that basic source code properties are provided with the final result, e.g. cyclomatic complexity (*Software metrics* mechanism).

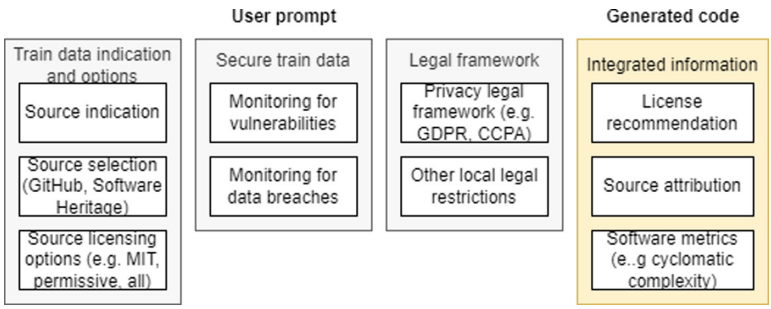

Fig. 1. Conceptual model for the consideration of software reuse properties in Generative AI tools.

5 Conclusions

In this paper, we have presented an overview of available GenAI tools for code generation and have presented a comparative view for a front-end use case that entails licensing and data privacy concerns. We have also discussed various uses in the literature and software reuse implications, concluding to a conceptual model with mechanisms that can assist in the consideration of licenses and data privacy. As future work, we will explore the functionality of additional LLMs for source code generation. We also intend to integrate some of the mechanisms

of the conceptual model in an open source Generative AI tool with a focus on license compliance. Future work could also examine other software reuse issues, such as creativity and avoidance of duplicated code.

References

1. Acher, M., Martinez, J.: Generative ai for reengineering variants into software product lines: an experience report. In: Proceedings of the 27th ACM International Systems and Software Product Line Conference-Volume B, pp. 57–66 (2023)
2. Ahmad, A., Waseem, M., Liang, P., Fahmideh, M., Aktar, M.S., Mikkonen, T.: Towards human-bot collaborative software architecting with chatgpt. In: Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering, pp. 279–285 (2023)
3. Corso, V., Mariani, L., Micucci, D., Riganelli, O.: Generating java methods: An empirical assessment of four ai-based code assistants. arXiv preprint arXiv:2402.08431 (2024)
4. Destefanis, G., Bartolucci, S., Ortu, M.: A preliminary analysis on the code generation capabilities of gpt-3.5 and bard ai models for java functions. arXiv preprint arXiv:2305.09402 (2023)
5. Ebert, C., Louridas, P.: Generative ai for software practitioners. IEEE Softw. **40**(4), 30–38 (2023)
6. Gottlander, J., Khademi, T.: The effects of ai assisted programming in software engineering (2023)
7. Haque, M.A., Li, S.: The potential use of chatgpt for debugging and bug fixing. EAI Endorsed Trans. AI Robot. **2**(1), e4–e4 (2023)
8. Idrisov, B., Schlippe, T.: Program code generation with generative ais. Algorithms **17**(2), 62 (2024)
9. Imai, S.: Is github copilot a substitute for human pair-programming? an empirical study. In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, pp. 319–321 (2022)
10. Jaeger, M.C., et al.: The fossology project: 10 years of license scanning. IFOSS L. Rev. **9**, 9 (2017)
11. Kapitsaki, G.M., Charalambous, G.: Modeling and recommending open source licenses with findosslicense. IEEE Trans. Software Eng. **47**(5), 919–935 (2019)
12. Kapitsaki, G.M., Tselikas, N.D., Foukarakis, I.E.: An insight into license tools for open source software systems. J. Syst. Softw. **102**, 72–87 (2015)
13. Liu, J., Xia, C.S., Wang, Y., Zhang, L.: Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. Adv. Neural Inform. Processing Syst. **36** (2024)
14. Mangini, V., Tal, I., Moldovan, A.N.: An empirical study on the impact of gdpr and right to be forgotten-organisations and users perspective. In: Proceedings of the 15th International Conference on Availability, Reliability and Security, pp. 1–9 (2020)
15. Moraes, J.P., Polato, I., Wiese, I., Saraiva, F., Pinto, G.: From one to hundreds: multi-licensing in the javascript ecosystem. Empir. Softw. Eng. **26**, 1–29 (2021)
16. Nguyen, N., Nadi, S.: An empirical evaluation of github copilot's code suggestions. In: Proceedings of the 19th International Conference on Mining Software Repositories, pp. 1–5 (2022)

17. Nguyen-Duc, A., et al.: Generative artificial intelligence for software engineering–a research agenda. arXiv preprint arXiv:2310.18648 (2023)
18. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R.: Asleep at the keyboard? assessing the security of github copilot's code contributions. In: 2022 IEEE Symposium on Security and Privacy (SP), pp. 754–768. IEEE (2022)
19. Reddy, H.R.: Jacobsen v. katzer: the federal circuit weighs in on the enforceability of free and open source software licenses. Berkeley Tech. LJ **24**, 299 (2009)
20. Roumeliotis, K.I., Tselikas, N.D.: Chatgpt and open-ai models: a preliminary review. Future Internet **15**(6), 192 (2023)
21. Sætra, H.S.: Generative ai: Here to stay, but for good? Technol. Soc. **75**, 102372 (2023)
22. Sobania, D., Briesch, M., Hanna, C., Petke, J.: An analysis of the automatic bug fixing performance of chatgpt. In: 2023 IEEE/ACM International Workshop on Automated Program Repair (APR), pp. 23–30. IEEE (2023)
23. Yetiştiren, B., Özsoy, I., Ayerdem, M., Tüzün, E.: Evaluating the code quality of ai-assisted code generation tools: an empirical study on github copilot, amazon codewhisperer, and chatgpt. arXiv preprint arXiv:2304.10778 (2023)

Variability and Reuse

Complexity of In-Code Variability: Emergence of Detachable Decorators

Jakub Perdek(✉) and Valentino Vranić

Institute of Informatics, Information Systems and Software Engineering Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, Bratislava, Slovakia
{jakub.perdek,vranic}@stuba.sk

Abstract. This paper presents a study on how selected approaches to expressing variability in code affect code complexity. To evaluate and compare the complexity of essential aspects of different approaches to in-code variability management, we designed five prominent cases of how variability is expressed in code: using decorators, using decorators without variability configuration expressions, using wrappers, using decorators with additional unwanted dead code constructs not being included for illegal decorators, and with no variability expressed in code at all. To measure code complexity in these cases, we used a framework for evaluating TypeScript code that we implemented in Java. Our framework is capable of assessing 15 metrics, comprising several variants of the LOC, Halstead, cyclomatic complexity, and cyclomatic density metrics. Decorators are detachable because they are decorating particular code construct with a predefined naming convention and no effects on code. The study was conducted on a software product line aimed at graphical applications we developed for evaluation purposes. We came to a range of interesting findings, such as that the detachable decorator version of introduced annotations is significantly less complex than other ways of expressing variability in code, annotations in comments (as in pure::variants), and tags (as in frame technology) do not directly affect the complexity of business functionality, decorators can be entirely separated from business logic, etc.

Keywords: code complexity · software product lines · variability management · decorators · variability configuration · comparative analysis

1 Introduction

Variability management is the central part of software product lines. Its in-code realization may significantly increase code complexity [5,9,24]. The configuration of products in software product lines is commonly managed with transformations or with visual languages [24]. Consequently, the exponential increase [27] of variants by the number of features complicates manual corrections [3], even when

A. Achilleos et al. (Eds.): ICSR 2024, LNCS 14614, pp. 51–71, 2024.
https://doi.org/10.1007/978-3-031-66459-5_4

resolving conflicts while incorporating new features into an existing software product line [1]. These corrections and conflicts should be resolved automatically. However, the complexity of in-code variability management constructs is neither evaluated, nor automatically applied to optimize configuration expressions. Its because they are used in comments or configuration variables are interleaved with business logic code.

This paper presents a study on how selected approaches to expressing variability in code affect code complexity. It is organized as follows. Section 2 presents some approaches to dealing with the complexity of expressing variability in code. Section 3 explains how we designed our study on how selected approaches to expressing variability in code affect code complexity, Sect. 4 explains how the study was performed and presents its results. Section 5 discusses the results. Section 6 relates this study to what others have done. Section 7 concludes the paper.

2 Variability Management Constructs and Code Complexity

Variability management is preserved in code mainly through wrapped comments such as in pure::variants [23]. An example is shown in Listing 1.1. Code fragments that implement variable features are being marked using annotations manually or in an automated fashion to allow their tracing and composition [5].

In addition to comments, tags can be used to express code templates as in the frame technology [17,18]. This requires dedicated transformation tools to be applied before the actual compilation. Another option to manage variability in code is to use conditional compilation [6], which also relies on tags (preprocessor directives). However, these approaches lead to polluting the code with tags [24], making it more complex.

```
1 //PV:IFCOND(pv:hasFeature(HazardWarning))
2 static int warning_lights_value; [REMAINING CODE OF HazardWarning...]
3 //PV:ENDCOND
```

Listing 1.1. Expressing in-code variability in pure::variants (adopted from pure::systems [23]).

Variability management based on aspect-oriented programming to some extent resolved evolvability, modularity, and code pollution issues [6], but remained not fully supported. Furthermore, aspects leave the affected code oblivious of its effects [7]. Variability configuring variables scattered through various aspects are usually used and mixed with the rest of the business code.

Annotations are usually realized with comments which require additional preprocessing tools [5]. It's because of the need for an easy establishment of annotation-based. [5] software product lines, technology support, and the mentioned restrictions. The code fragments that implement variable features are wrapped by predefined comments that are later processed to allow for their tracing.

Complexity metrics have been successfully applied to measure the difficulty of code comprehension [11]. However, the comments usually do not contribute to this process because they are omitted during compilation. Under such circumstances, evaluating the complexity of these comments and related quality metrics is impossible. If aspect-oriented programming is used, conditions can be expressed directly by variables in aspects. In such case, separating and measuring the complexity of particular constructs is challenging because variables that guard variability conditions according to configuration tend to be mixed with business functionality.

Configuration expressions are formulas used to decide whether to include or exclude processed code fragments if conditions are fulfilled. They can contain additional information associated with an annotated variation point. Analysis of the impact of their hierarchically expressed version [21] is required for flexible configuration in a particular context. An example of such an expression is in Listing 1.4.

```
1 // @ts-ignore
2 @DecorSRVC.skipLVP({"algoType": "[’
  A1’, ’A2’, ’A3’]"}, "import { State
   } from ’../store’;") var newA;
3 //import { State } from ’../store’;
```

Listing 1.2. Annotated import statement by decorators.

```
1 @DecorSRVC.wBlock({"zoom": "true"})
2 public zoom(zoomHTML: any): void {
3     //DO ZOOMING
4 }
```

Listing 1.4. Annotated method by decorators.

```
1 var EXP_START6 = { "algoType": "[’
  A1’, ’A2’, ’A3’]" };
2 import { State } from "../store";
3 var EXP_END6 = { "EXP_END": "--" };
```

Listing 1.3. Wrapped code of the import statement.

```
1 var EXP_START0 = {"zoom": "true"};
2 public zoom(zoomHTML: any): void {
3     //DO ZOOMING
4 }
5 var EXP_END0 = { "EXP_END": "--" };
```

Listing 1.5. Wrapped code of the method.

3 Designing the Study

In the study reported in this paper, the code complexity evaluation of essential aspects of variability management was performed on adaptation of annotations in comments—as they are used in pure::variants—, preprocessor directives—as they are used in conditional compilation—, and tags—as they are used in frame technology— with the code construct equivalents we proposed. The type and use of annotations are taken from our method of developing lightweight aspect-oriented software product lines with automated product derivation [21]. Similarly, we consider the complexity of used configuration expressions and overhead caused by necessary dead code. We adapted programming language implementations of decorator pattern to design decorator-bound form which is perceived as

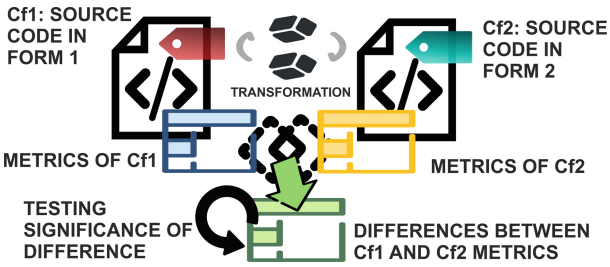

Fig. 1. Comparative analysis of code constructs complexities of variability management.

the cleanest and most adaptable to us. The possibility of transforming it into different variants enables flexible adaptation of conventional versions of annotations in source-code comments. This form with unsupported decorators is converted into a compilable version with supported constructs that can be evaluated and used further in its less concise structure. In TypeScript this is caused by unavailable types of decorators or illegal decorators [28]. Many tools cannot process the latter, but these constructs remain in the generated abstract syntax trees. Associated configuration expressions can be optionally excluded to evaluate their complexity in a particular context and improve them accordingly.

All mentioned challenges with measuring and comparing code complexity are solved with our framework. Specifically, the code complexity from each file is transformed into a pair of variability management versions for analysis and visualized with code itself, evaluated complexities, differences of evaluated complexities, and a statistic test performed to measure if this pair of variability management versions/cases significantly differ. We propose a process to compare two annotated files with variability management fully realized in code as shown in Fig. 1. This process is supported by a framework, which is available on Github,[1] along with all scripts, software product line aimed at graphical applications we developed for evaluation purposes, and resulting artifacts.

3.1 Adapting the Wrappers and Designing the Detachable Decorators

We designed wrappers equivalent to available variability management constructs based on conditional compilation constructs, variability comments as in pure::variants, and tags as in the frame technology (either pure or supported by aspect-oriented programming). Our wrappers are a part of the syntax of the programming language being used for software product line development (TypeScript in our case). They are active code constructs, i.e., they are not preprocessor directives, tags, nor comments.

[1] https://github.com/jperdek/variabilityMgmtCodeConstructsComplexity.

Wrapping happens for the array of members or statements in the final abstract syntax tree by putting variability management elements before and after each selected sequence of members or statements. Additionally, an alternative "else" branch can optionally extend the wrapped functionality and help handle situations where given features are unavailable for particular configurations. Compared to decorators, nested wrappers are usually more difficult to read.

Contrary to this, we are introducing an adjustment of decorators for variability management taking into account their limited support to annotate only function parameters, classes, class methods, and variables [2]. Only these annotable variable units at the code level are prioritized in representing variability. Accordingly, decorators are proposed as a more adaptable form for managing variability and evaluating complexity. They can be associated with modular source code structures in a more comprehensible way. Similarly, they affect code execution only in cases of providing support for dynamic variability management. They are fully distinguished from the rest of the business logic (its constructs) by the predefined names, which makes them detachable and independent of variability management. Compared to decorators, the wrappers is highly applicable, but cannot be restricted or bound to particular code structures. The difference can be seen by comparing Listing 1.2 to Listing 1.3 and Listing 1.4 to Listing 1.5. In all listings, the same business logic is denoted as variability. Specifically, the import statement occurs in the first pair and the zoom method in the second. The first member of the pair consists of a decorator beginning with @, and the second is a conventional wrapper bounded with two initialized variables. Listings 1.2 and 1.4 employ a decorator beginning with @, while the other two listings employ a conventional wrapper bounded with two initialized variables.

Other forms can be based on constructs, such as if statements or for loops, but their use directly interleaves with code and possibly negatively affects modularity.

Each of the mentioned forms needs developers to be aware of the product derivation mechanism to the depth necessary to know its capabilities. For example, when code is annotated in wrong places, expressed with an unknown character sequence, or in the case of templates, it causes additional changes to the previous development style.

A particular product derivation mechanism usually does not demand preserving code modularity and extendability when variability needs to be handled at the code level. Consequently, if recognizable constructs for this mechanism are not preserved in source code, the mechanism cannot manage and synchronize how available code constructs are used. Extending their semantics to variability management increases overall complexity, especially if conditional compilation is used. In particular, the statements that express variability cannot be easily semantically distinguished from those statements that express business logic.

To evaluate and compare the complexity of essential aspects of different approaches to in-code variability management, we designed the following cases:

Case 1. Variability is expressed using detachable decorators

Case 2. Variability is expressed using detachable decorators, but without variability configuration expressions
Case 3. Variability is expressed using wrappers
Case 4. Variability is not expressed at all
Case 5. Variability is expressed using detachable decorators, but additional unwanted dead code constructs are not included for illegal decorators

Each case essentially differs in one or more characteristics, including the availability and format of configuration expressions as inner members (JSON or attributes), the way code constructs are used for variability management (wrapping, annotating, or none), the type of used constructs for variability management (decorators, variables, or preprocessor directives), and the necessity to use dead code for visual adaptation of variability management code constructs to places affected by variability. We did not consider conditional compilation with associated preprocessor directives because such representation interleaves with business logic code. We analyzed only configuration expressions in the JSON format due to the possibility of directly analyzing them as JavaScript/TypeScript objects, modeling and preserving hierarchic relations with feature models in code, and using a more concise format over conditions consisting of variables that usually interleave with business logic code.

Despite the mentioned characteristics, we must also consider available code constructs in a given programming language, capabilities of code complexity evaluation tools, and options to position available code constructs near target places visually. The comparative analysis proposed here was realized in scenarios each of which compares two of the in-code variability management cases we designed.

3.2 Hypotheses and the Process

We propose the following hypotheses to evaluate the effect of in-code variability constructs on code complexity, namely cyclomatic complexity, LOC, and Halstead measures:

Hypothesis 1. Variability expressions extracted from annotations do not significantly change the complexities of most evaluated metrics.
Hypothesis 2. Changing from wrappers to detachable decorators significantly improves the complexity of most evaluated complexity metrics.
Hypothesis 3. Removal of all variability constructs from Case 1 does not significantly change at least one of the evaluated complexity metrics.
Hypothesis 4. Unwanted dead code constructs significantly change complexity measured by most evaluated complexity metrics.

Validating Hypothesis 1 involves evaluating the complexity of configuration expressions represented in JSON format [21] with all mentioned metrics to show how code complexity is increased in a particular context. Specifically, Case 1 and 2 are compared. We assume that the complexity of configuration expressions significantly affects particular complexity metrics. Minor optimizations to

complexity evaluation can be performed by putting JSON into a string so that it forms one expression and measuring the final effect. This change will comprise the isolation of variability configuration expressions during measurements from the rest of the code, especially from business logic.

In future, such expressions could be collected, updated, compared with other expressions, and possibly optimized in an automated process. Making configuration expressions less complex and more comprehensible can help developers quickly learn relations from automatically generated semantic structural views [22] of an annotation-based software product line [5] (for example, with our matrix-based approach to structural and semantic analysis in software product line evolution [22]).

Configuration expressions influence the code only marginally if these differences are insignificant. Finally, the measured effect can be used further to design more optimal expressions to suit various extrafunctional requirements in an automated way.

Validating Hypothesis 2 involves evaluating the code complexity measures of the wrappers (based on already used wrappers in the form of comments) with detachable decorators (modern decorators) for variability management. Some of their negative effects on code complexity are already known [21]. Firstly, the complexity is increased in two places simultaneously when the content is bounded. In an abstract syntax tree of a TypeScript program, wrapping a particular element is possible only by adding neighbors before and after in an array of statements or members. Secondly, the nested code is hard to read when code-variability constructs must be paired. The information about how precisely quality is changed for different metrics is evaluated by comparing the complexity of wrappers (Case 3) with the one based entirely on detachable decorators (Case 1).

Validating Hypothesis 3 involves evaluating how complex variability constructs are in the particular context. In particular, we intend to find at least one metric that is significantly unaffected by variability management constructs. Such measurements are achieved by removing all variability constructs (Case 4) and comparing them to the original software product line with variability management (Case 1 or 3). The corresponding functionality can still be grouped according to used configuration expressions and assessed separately. Identification of additional complexity after introducing variability management is a basis for distinguishing how complex this functionality is.

Validating Hypothesis 4 involves evaluating the effects of legalizing some of the currently illegal detachable decorators on the code complexity of variability management. In the case of visual relatedness caused by the positioning of these variability constructs close to variation points, their syntactic representation is not connected with their visual one. Additionally, their transformation into another should be considered to evaluate approximated complexity. None of the available detachable decorators support an ideal form that allows for annotating various one-line code fragments. Even illegal detachable decorators used to annotate a particular variable require another dead code construct to be positioned near the target place to cover a variable line of code, such as an import

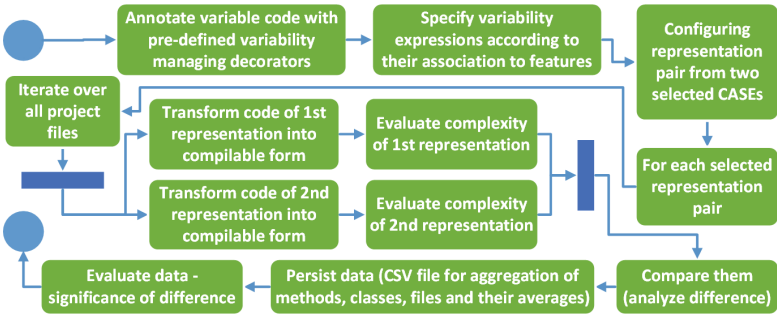

Fig. 2. Validating the hypotheses.

(see Listing 1.2) or call of a particular functionality. Transformation into the case with all supported detachable decorators (Case 1) is required to observe effects on quality, especially to source code complexity in a given context. Consequently, dead code constructs affect final complexity and can be preserved during transformation. In addition, the complexity of used detachable decorators can be approximated from average values of measurements taken previously or simply the compilable wrappers to which the code is transformed is used. We applied the latter in our study.

The process of validating the proposed hypotheses is displayed in Fig. 2. The base/actual case is transformed into a new one, followed by code complexity measurements. Finally, the subtraction of complexities determines the complexity of extended or different functionality. Firstly, variability configuration expressions are put into adapted decorator or wrapper code constructs to cover various variable code fragments. Secondly, project files are loaded and transformed into each pair of particular cases. Thirdly, the complexity is evaluated for each member with their difference in the separate measurements. Transformed scripts used to measure complexity are optionally persisted into files. Finally, data are analyzed according to chosen hypotheses and possible use cases.

4 Performing the Study

New code constructs in a particular programming language or their visual positioning in the code allow for variability management constructs to appear cleaner. For example, decorators in TypeScript can be customized with their own easily recognizable name and used to automatically transform these recognized marked points (variation points) into wrappers or even any evaluable forms. Additionally, they can be transformed into constructs that are impossible to distinguish automatically or where this process is error-prone. Their flexibility can be compared according to the code complexity of such variability markers or even associated configuration expressions used in code under a given context, such as annotating certain classes or files in a software product line as variable.

Fig. 3. Pair comparison of evaluated particular complexity metric on variability annotated files among all introduced cases: Part 1

Considering them as code constructs, their effect on the resulting code complexity can be evaluated similarly to that of a variability-unaffected code. In our study, we omit variability-unaffected files and all their code and instead focused on standalone files, their imports, classes, and various types of methods. Specifically, focus on files as standalone units is essential for considering the variability of import statements. In case merging all files together will cause not correct evaluation of overall complexity. Thanks to the TyphonJS-ESComplex service [15], source code complexity metrics are evaluated, considering decorator complexities. Supported ones are cyclomatic complexity (cyclomatic number and cyclomatic density), Halstead measures (Bugs, Difficulty, Effort, Length, Time, Vocabulary, Volume, information about distinct identifiers such as Operands and Operators), and number of lines of code (LOC). Maintainability is also evaluated for some code structures, such as classes.

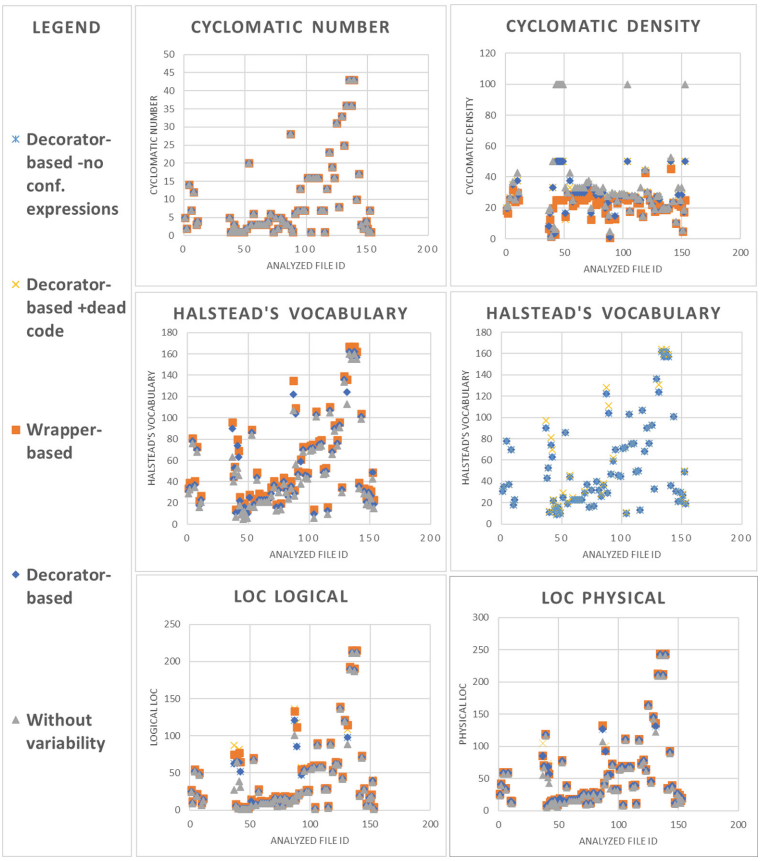

Fig. 4. Pair comparison of evaluated particular complexity metric on variability annotated files among all introduced cases: Part 2

As has been mentioned in Sect. 3, the study was performed on a software product line aimed at graphical applications, which we developed for evaluation purposes.

We iterated with the design of annotations (detachable decorators) with available decorators in TypeScript to cover all variable parts more concisely. One-line declarations, calls, imports, and other code fragments showed to be problematic to decorate. Most of these code fragments cannot be wrapped inside functions. Additionally, it's not feasible to study complexity due to unavailable tools supporting illegal decorators in TypeScript. Despite this, we used illegal experimental decorators that annotated newly declared variables and put them under one-line variable code fragment that needed to be covered. In the next step, they are transformed into a wrappers that can be compiled, especially for the evaluation phase. For some cases, such as in Angular, the compiler can ignore

the illegal use of decorator with @ts-ignore, thus allowing native development of applications. The code is still slightly polluted with necessary dead code.

For each target project file, we created a hierarchic structure from metrics being assessed with the possibility of applying various operations, such as a difference between two such structures. Finally, results are divided into files, classes, their methods, and averages when many classes or methods are inside of given files. Due to the domain orientation/solution design, the results strongly depend on the given variability markings and the technology and frameworks used. A different domain or solution design can require a different number of annotations that can vary in their various types. For this purpose, the analysis is applied to files of software product line rather than on their merged version and then statistically evaluated. In addition, the resulting complexities are divided by the number of decorators used. Consequently, the number and complexity of evaluated code constructs should not significantly differ from other associated and similarly analyzed files after transformation is applied into a different form. Still, the more use cases are implemented, the more complex variability expressions and more dense annotations are, especially for contradicting features. Accordingly, the resulting complexities are also affected. These initial conditions do not prevent observing variability management's additive effect on code complexity. Additionally, various code-based variability annotations are compared based on their complexities and evaluated to see if unsupported illegal decorators or their future versions will improve complexity and quality.

Different complexity metrics extracted from the code transformed according to proposed cases are quantitative, except a list of used identifiers and operators in the Halstead measures. Additionally, performed normality tests do not confirm normality; thus, a test independent of probability distribution is chosen. After getting information about possible correlations, all proposed metrics were shown to be dependent. It also holds for measurements that compare the code in a particular case (sample) and its transformed/extended version (opposite sample). Thus, pair testing is used. Whether compared complexity measures significantly differ after and before applying the particular case of variability management annotations helps to find unaffected measures or those with marginal values and is verified with the paired nonparametric Wilcoxon test. If the significance level or p-value is below the given threshold of 0.05, the zero hypothesis is rejected, and the samples are significantly different. Another option is to use a Kruskal-Wallis test, which uses the Nemenyi method to determine if a change is also significant.

In this experiment, over 76 variability annotated files that contain at least one variability annotation are tested. Additionally, the 84 classes stored in the mentioned files were used to test their in-class complexities, but only 64 were unique. The differences between each file/member pair cases (in each vertical line of the graph) are visualized in Figs. 3 and 4.

4.1 Code Complexity of Variability Configuration Expressions

Our primary focus was to evaluate whether changes to different complexity measures are significant after inserting variability configuration expressions. Quantitative results also help optionally optimize used constructs or recommend their use. The solution with configuration expressions seems more complex than the version without them. Halstead measures, such as Bugs and Length, always increased after introducing configuration expressions. Halstead's Vocabulary and Identifiers, such as operands and operators, always increased or remained unchanged. They remained unchanged due to using empty JSON in cases where the whole file should be copied according to our variability management mechanism policy. The logical LOC has a similar tendency. The cyclomatic complexity and the number of physical LOC remained unchanged. Halstead's Difficulty, Effort, and Time are decreased or increased for some measures.

Table 1. Code complexity for Case 1 and 2 compared.

Name of compared metric	Corr.	W	p-value	95% CI	Est.	p>0.05
Cyclomatic Complexity	1.0000	0	1.0000E+00	NaN, NaN	NaN	TRUE
Cyclomatic Density	0.9897	0	2.1335E-04	−2.12, −0.7690	−1.4110	FALSE
Halstead's Bugs	0.9989	2926	1.9956E-14	0.002, 0.005	0.002	FALSE
Halstead's Difficulty	0.9976	294	3.8958E-02	0.07, 0.59	0.3265	FALSE
Halstead's Effort	0.9992	2652	7.5836E-10	57, 103	79.7495	FALSE
Halstead's Length	0.9988	2926	1.2543E-15	1.00, 2.50	1.0001	FALSE
Halstead's Time	0.9992	2652	7.5836E-10	3.19, 5.74	4.4308	FALSE
Halstead's Vocabulary	0.9990	435	1.9687E-06	1.50, 3.50	2.0000	FALSE
Halstead's Volume	0.9903	2926	3.6708E-14	35.18, 45.23	38.6940	FALSE
Halstead's Id Dist. Operands	0.9984	171	1.3795E-04	2.00, 4.00	3.5001	FALSE
Halstead's Id Ttl Operators	0.9981	171	1.5794E-04	2.00, 13.50	9.0000	FALSE
Halstead's Id Dist. Operators	0.9993	66	1.0893E-03	NaN, NaN	1.0000	FALSE
Halstead's Id Ttl Operators	0.9982	2926	1.2460E-15	1.00, 1.50	1.0001	FALSE
LOC Physical	1.0000	0	NaN	NaN, NaN	NaN	TRUE
LOC Logical	0.9978	171	1.5853E-04	1.00, 7.00	5.0000	FALSE

We tested the significance of these changes in the paired test on a significance level of 0.05. The results are shown in Table 1. Only the cyclomatic complexity and number of logical LOC do not deny Hypothesis 0, thus confirming that these measures are not significantly different except the remaining ones. Accordingly, our Hypothesis 1 is rejected. Evaluating complexities from classes showed that Halstead's Difficulty is near the boundary of being significantly different and can be improved by removing redundant classes used only to annotate and preserve given files during variability handling. Class maintainability is also evaluated and remains without significant difference in this case.

4.2 Wrappers Vs. Detachable Decorators

The similarly applied paired Wilcoxon test on Case 3 and 1 pair proved that the wrappers and detachable decorators significantly differ in all complexity metrics except the cyclomatic complexity. The remaining code fragments are more complex for wrappers by reaching higher values for some Halstead complexity measures (Bugs, Length, Operator and Operand identifiers, Volume, and Vocabulary, except one sample). LOC (physical and logical) followed this tendency despite a few similar values. Not always, but complexity is high for other Halstead measures (Difficulty, Effort, and Time). Declaring new variables is affected by the complexity of the whole file and results in different complexities. Halstead's Bugs, Length, and Volume are mainly increased when their content is wrapped inside a method and then when classes are wrapped. These preferences are for Vocabulary usually swapped. The simpler the wrapped code is, the lower the differences are. Only cyclomatic density is decreased.

Table 2. Code complexity for Case 3 and 1 compared.

Name of compared metric	Corr.	W	p-value	95% CI	Est.	$p > 0.05$
Cyclomatic Complexity	1.0000	0	1.0000E+00	NaN, NaN	NaN	TRUE
Cyclomatic Density	0.8226	0	3.5776E-13	−4.1959, −2.28	−3.02	**FALSE**
Halstead's Bugs	0.9997	2556	2.4526E-13	0.01, 0.02	0.0141	**FALSE**
Halstead's Difficulty	0.9971	2237	5.9298E-09	0.60, 0.80	0.7390	**FALSE**
Halstead's Effort	0.9988	2386	2.2106E-10	503.04, 1493.10	841.6983	**FALSE**
Halstead's Length	0.9997	2556	9.3382E-17	6.00, 6.00	6.0000	**FALSE**
Halstead's Time	0.9988	2386	2.2106E-10	27.95, 82.95	46.7609	**FALSE**
Halstead's Vocabulary	0.9994	2484	4.2116E-14	3.00, 3.45	3.0000	**FALSE**
Halstead's Volume	0.9997	2556	2.4761E-13	39.32, 45.96	42.2363	**FALSE**
Halstead's Id Dist. Operands	0.9996	2415	2.5713E-16	2.00, 2.00	2.0000	**FALSE**
Halstead's Id Ttl Operands	0.9999	2556	9.3382E-17	2.00, 2.00	2.0000	**FALSE**
Halstead's Id Dist. Operators	0.9886	2030	2.1410E-12	1.00, 1.50	1.0000	**FALSE**
Halstead's Id Ttl Operators	0.9999	2485	9.8502E-17	4.00, 4.00	4.0000	**FALSE**
LOC Physical	0.9998	2556	2.1563E-16	1.00, 1.00	1.0000	**FALSE**
LOC Logical	0.9999	2485	9.8502E-17	2.00, 2.00	2.0000	**FALSE**

The Wilcoxon test confirmed the significance of these differences in all cases except the cyclomatic complexity. In conclusion, Hypothesis 2 cannot be rejected. Consequentially, their significance level is too far from the threshold of 0.05, which makes the detachable decorators (Case 1) preferable over the wrappers (Case 3). Additionally, the full support of decorators for one-line constructs will probably improve results when they are used during comparison with wrappers and deepen the differences between both versions. The cyclomatic complexity is not affected by any change inside a given file caused by the variability management's independence. Results are displayed in Table 2.

4.3 Significance of Variability Management Code Complexity

Hypothesis 3 is proposed to test the significance of the complexity of various variability constructs from Case 1 to each overall business code complexity (Case 4) for each file. The improvements are presented in Table 3 and showed that detachable decorators strongly affect the majority of complexity measures except for cyclomatic complexity (no change) and Halstead's Difficulty. Some complexity metrics contribute in both directions, especially some Halstead measures (Difficulty, Effort, and Time) measured for each analyzed file. Cyclomatic density is the only one that decreased after the variable code fragments were removed.

Table 3. Code complexity for Case 1 and 4 compared.

Name of compared metric	Corr.	W	p-value	95% CI	Est.	$p > 0.05$
Cyclomatic Complexity	1.0000	0	NaN	NaN, NaN	NaN	TRUE
Cyclomatic Density	0.9264	0	3.6200E-14	−4.63, −2.05)	−2.9825	FALSE
Halstead's Bugs	0.9900	2926	3.6200E-14	0.01, 0.02	0.0130	FALSE
Halstead's Difficulty	0.9921	1489	8.9495E-01	−0.19, 0.52	0.0280	**TRUE**
Halstead's Effort	0.9920	2486	1.2000E-07	123.31, 199.05	155.8794	FALSE
Halstead's Length	0.9885	2926	1.2500E-15	5.00, 6.50	5.0001	FALSE
Halstead's Time	0.9920	2486	1.2000E-07	6.85, 11.06	8.6572	FALSE
Halstead's Vocabulary	0.9880	2926	1.2900E-14	3.00, 3.50	3.0000	FALSE
Halstead's Volume	0.9903	2926	3.6700E-14	35.18, 45.23	38.6939	FALSE
Halstead's Id Dist. Operands	0.9855	2926	1.2500E-15	2.00, 3.00	2.0001	FALSE
Halstead's Id Ttl Operands	0.9891	2926	1.2500E-15	2.00, 3.00	2.0001	FALSE
Halstead's Id Dist. Operators	0.9928	406	2.6600E-06	1.50, 2.00	1.9999	FALSE
Halstead's Id Ttl Operators	0.9863	2926	1.2500E-15	3.00, 3.50	3.0001	FALSE
LOC Physical	0.9904	2926	1.0300E-16	2.00, 2.00	2.0000	FALSE
LOC Logical	0.9734	2926	1.2500E-15	1.00, 1.50	1.0001	FALSE

The wrapped code significantly affects (from half to more than twice for the one annotated code fragment, but more than twice for the standalone file) Halstead's Bugs, Length, Vocabulary, Operator and Operand identifiers, and Volume. Both the logical and physical number of LOCs are minimally doubled. In the case of Halstead's Bugs, the lowest contributions are associated with statements only used to support variability, as mentioned before. Consequently, using supporting code for variability management can be limited, and illegal detachable decorators seem preferable to wrappers. The most affected are modules that should be configured differently to reduce additional variables, "if", "than", "else" constructs, and necessarily associated import statements. The last one is the most important to handle because of the necessity to use wrappers; otherwise, they must be held at least with illegal detachable decorators. Their nature is thus not intended directly to handle variability.

Additionally, the removal of files with the most wrappers decreases most Halstead measures. The test on significance between the state before and after applying variability management was performed again, and results are displayed in Table 4. The significance level improved in most cases by about a hundredfold (expressed in bold italics) and tenfold (bold) for the remaining ones, except for cyclomatic complexity and physical LOC. This test showed how reducing wrapper parts in cases such as module and routing configuration or services with additional code handling in our software product line aimed at graphical applications can help reduce the code complexity of used variability management constructs to mark variable code. Still, the change is too far from the threshold of 0.05. In summary, using variability management constructs significantly affects most used complexity measures and cannot decrease them by any available code construct below the significance threshold. Hypothesis 3 is not rejected, thanks to Halstead's Difficulty. Still, less complex variability-aware code with higher quality can be produced in this restricted way.

4.4 The Effect of Dead Code Constructs on Code Complexity

The next step is to discover how redundant dead code affects the analyzed complexity measures in order to validate Hypothesis 4. This code is used in helper functionality based on illegal decorators in TypeScript that mediate visually positioning variability annotations next to the variable code fragments as an alternative to wrapper constructs. The solution required using these constructs in only 9 out of all 76 files. Subtracting each of the complexity measures of Case 5

Table 4. Code complexity for Case 1 and 4 compared without most of the files with wrappers.

Name of compared metric	Corr.	W	p-value	95% CI	Est.	p > 0.05
Cyclomatic Complexity	1.0000	0	NaN	NaN, NaN	NaN	TRUE
Cyclomatic Density	0.9277	0	***1.6427E-12***	−4.58, −1.81	−2.7695	FALSE
Halstead's Bugs	0.9969	2211	***1.6442E-12***	0.01, 0.01	0.0120	FALSE
Halstead's Difficulty	0.9948	886	**1.6171E-01**	−0.28, 0.11	−0.1545	TRUE
Halstead's Effort	0.9979	1787	***1.3588E-05***	102.09, 152.93	126.98	FALSE
Halstead's Length	0.9965	2211	**2.5043E-14**	5.00, 5.00	5.0000	FALSE
Halstead's Time	0.9979	1787	***1.3588E-05***	5.67, 8.50	7.0545	FALSE
Halstead's Vocabulary	0.9963	2211	**4.1183E-13**	2.50, 3.50	2.9999	FALSE
Halstead's Volume	0.9969	2211	***1.6743E-12***	32.77, 38.41	35.4739	FALSE
Halstead's Id Dist. Operands	0.9953	2211	**2.5043E-14**	2.00, 2.00	2.0001	FALSE
Halstead's Id. Ttl Operands	0.9954	2211	**2.5043E-14**	2.00, 2.00	2.0001	FALSE
Halstead's Id. Dist. Operators	0.9958	171	***1.4868E-04***	2.00, 3.00	2.0000	FALSE
Halstead's Id. Ttl Operators	0.9953	2211	**2.5043E-14**	3.00, 3.00	3.0001	FALSE
LOC Physical	0.9980	2211	7.4931E-16	2.00, 2.00	2.0000	FALSE
LOC Logical	0.9958	2211	**2.5043E-14**	1.00, 1.00	1.0001	FALSE

with unwanted helper code from the case without them measures their overhead. Complexity is again increased in most Halstead metrics (Bugs, Effort, Length, Time, Vocabulary, and Volume) and LOC metrics (logical and physical). Only cyclomatic density decreased in all cases. The results of the paired Wilcoxon test that tests the significance of the change for most of the complexity metrics are shown in Table 5. Hypothesis 4 cannot be rejected. Still, compared with previous measurements, the significance level is not far from the threshold of 0.05 for most evaluated complexity metrics.

Table 5. Code complexity for Case 5 and 1 compared.

Name of compared metric	Corr.	W	p-value	95% CI	Est.	$p > 0.05$
Cyclomatic Complexity	1.0000	0	1.0000	NaN, NaN	NaN	TRUE
Cyclomatic Density	1.0000	0	0.0092	−1.11, −0.25	−0.495	**FALSE**
Halstead's Bugs	0.9998	45	0.0092	0.01, 0.04	0.0205	**FALSE**
Halstead's Difficulty	0.9999	40	0.0440	0.01, 0.53	0.2210	**FALSE**
Halstead's Effort	0.9996	45	0.0092	551.81, 2876.1	1199	**FALSE**
Halstead's Length	0.9998	45	0.0091	4.00, 16.00	9.0001	**FALSE**
Halstead's Time	0.9996	45	0.0092	30.66, 159.78	66.603	**FALSE**
Halstead's Vocabulary	0.9999	45	0.0034	NaN, NaN	1.0000	**FALSE**
Halstead's Volume	0.9998	45	0.0092	32.04, 108.08	62.861	**FALSE**
Halstead's Id Dist. Operands	0.9999	45	0.0034	NaN, NaN	1.0000	**FALSE**
Halstead's Id Ttl Operands	0.9996	45	0.0091	2.00, 8.00	4.5000	**FALSE**
Halstead's Id Dist. Operator	1.0000	0	1.0000	NaN, NaN	NaN	TRUE
Halstead's Id Ttl Operators	0.9999	45	0.0091	2.00, 8.00	4.5000	**FALSE**
LOC Physical	0.9990	45	0.0092	2.50, 12.00	5.5000	**FALSE**
LOC Logical	0.9996	45	0.0091	2.00, 8.00	4.5000	**FALSE**

5 Discussion

The experiments depend on variability management policies, which prescribe how information about variability is marked. Our software product line aimed at graphical applications is based on our method of developing lightweight aspect-oriented software product lines with automated product derivation and its policies [21].

Additionally, the values of specific metrics can differ under different variability configurations according to the number and complexity of variable features. Only five features and five subfeatures were implemented, along with one annotation per file for the initial configuration of variability management, which proved significantly different from the original business code. Modern software product

lines count from tens to thousands of features [13], where many are scattered across multiple files and commits during evolution [10]. Consequently, this introduces numerous variability constructs with more complex configuration expressions. Managed modules in the Angular framework require annotating various fragments. On the contrary, after removing such cases, the result of the presented test is still closely behind the threshold to reject a zero hypothesis and proves a significant difference. Still, the complexities can be improved even with minor changes. Some Halstead metrics can lead to unclear properties for small or middle programs, possibly affecting their interpretation [29]. Consequently, the measured values should be verified with user testing, especially for variability management. Another option is to use them according to updated versions on a large scale.

Additional metrics should be introduced to fully observe the comprehension of particular variability management constructs, especially expressions by developers. Variability expressions are usually mapped to features positioned in hierarchic feature models or belonging to the chains of dependencies amongst features. These relations are not considered and pose the problem of constructional validity.

The detachable decorators even empirically overcomes the wrappers when the ending part should be necessarily included, the nesting of wrappers is error-prone and hard to read, and even they cannot be directly bound to a particular entity in code. All LOC and Halstead metrics agreed for conclusion validity, but complications emerged with cyclomatic complexity. Specifically, conditional reasoning is affected by variability conditions, but they do not affect cyclomatic complexity. This measure failed in all introduced cases due to a misunderstanding of the role of variability management constructs.

We observed that the complexity of detachable decorators as constructs bound to other code constructs depends on the complexity of these constructs. Consequently, we found the problem of internal validity because our constructs are easily detachable without any side effects and thus not bound to business logic with its flows. On the contrary, the more modular the annotated business entity is, the better these constructs are for modularization. Specifically, code-complexity metrics are unaware of variability management and pose the problem of inner validity. New metrics should be introduced for this purpose.

The problem of generalizing outcomes for constructs belonging to other programming languages or computation models under specific conditions falls into external validity. For example, transitions of finite automaton or state machine can be aggregated or grouped. Still, it is useless to annotate them individually if simulators in this process do not support advanced data structures. Unknown relations to variability or immature technologies cause these problems.

6 Related Work

The most similar expressions to configuration expressions are presence conditions [4]. They manage the presence of particular functionality in various models according to the configuration of features [4], usually through feature models.

The same operators (and/or) are used with configuration in variability management on various models. Their known drawback is the ability to manage the presence only of available functionality, negative variability [24]. Compared with our JSON [21] configuration expressions instantiated as objects in JavaScript/-TypeScript, a presence condition cannot easily express hierarchic information and even hold more information, such as additional strings or values assigned during development. If applied at the code level, the variables used in these presence conditions usually interleave with the code [21]. To the best of our knowledge, their code complexity was never assessed nor used to optimize them.

Many authors focused on short programs [12,19,20] possibly because of the lack of an automated support for their studies. However, we developed such support, which made possible for our study to be of a nontrivial size.

Athar Khan et al. compare three programming languages according to measured metrics on the implementation of a sorting algorithm [12]. LOC, cyclomatic complexity, and the Halstead measures were used to determine the most complex implementation and implications for testing time. Exceptionally for the Halstead difficulty, the different most complex candidate programming language was determined [12].

Peitek et al. observed the relation between metrics and brain activity [20]. The fMRI study [20] found only a small correlation of the LOC and Halstead measures with the response time and a medium correlation with the correctness of the resulting program outputs. The medium correlation also held for brain activation and deactivation, while cyclomatic complexity is not correlated in any case. The reason for using the Halstead measures in this study is given by the increase in cognitive processing demands due to the number of symbols [25]. Similarly, cyclomatic complexity was used to analyze control flows due to their effects on rule-guided conditional reasoning [14,16]. Our study could be extended to evaluate the level of cognitive processing and conditional reasoning in different approaches to variability management.

Sehgal and Mehrotra [26] used Halstead's volume to predict faults among system components in the reliability and mean time between failures metric, which considers usage time and the amount of code with operators and operands. Graves et al. [8] showed how complexity metrics correlate with each other in a similar way our study indicated.

7 Conclusions and Future Work

This paper presents a study on how selected approaches to expressing variability in code affect code complexity. To evaluate and compare the complexity of essential aspects of different approaches to in-code variability management, we designed five prominent cases of how variability is expressed in code: using decorators, using decorators without variability configuration expressions, using wrappers, using decorators with additional unwanted dead code constructs not being included for illegal decorators, and with no variability expressed in code at all. To measure code complexity in these cases, we used a framework for evaluating TypeScript code that we implemented in Java. Our framework is capable

of assessing 15 metrics, comprising several variants of the LOC, Halstead, cyclomatic complexity, and cyclomatic density metrics. Decorators are detachable because they are decorating a particular code construct with a predefined naming convention and have no effects on code. The study was conducted on a software product line aimed at graphical applications we developed for evaluation purposes.

We came to a range of interesting findings. Detachable decorators proved to be significantly less complex than other ways of expressing variability in code, especially than the in-code version of wrappers. Except for dynamic self-adaptability, annotations in comments (as in pure::variants) do not directly affect the complexity of business functionality. Similarly, decorators can be entirely separated from business logic and their names are easily adaptable to particular domains or extensions. They are even applicable to incorporating changes dynamically on the fly and their application is forced to improve modularity. Configuration expressions are put inside their arguments more concisely in the JSON format. On the contrary, illegal decorators are still necessary for configuration purposes or exceptional cases, rather than traditional wrappers to annotate import statements. Additionally, introducing redundant code for these purposes significantly affects complexity measures. However, even decorators proved significantly more complex for various metrics than not having variability expressed in code at all.

The configuration expressions used for their hierarchic nature to concisely express feature models significantly increase most code complexity measures, especially the Halstead ones. Consequently, optimization to more comprehensive versions and even to make easier creating autonomous processes of optimizing configuration expressions and evolving software product lines are required to configure, simulate, and evaluate effects on the extensive number of features. In our future work, the results can help design a tool to visualize only certain views of variability on the fly and update particular configuration expressions accordingly. Additionally, we want to automatically optimize configuration expressions and evaluate which hierarchically expressed configuration made according to feature model fits the best.

Acknowledgements. The first author was supported by the STU Grant Scheme for Support of Young Researchers.

References

1. Blair, L., Pang, J.: Aspect-oriented solutions to feature interaction concerns using AspectJ, p. 17 (2003)
2. Cardoso, J.: How to use decorators in TypeScript. digitalocean.com (2021). https://www.digitalocean.com/community/tutorials/how-to-use-decorators-in-typescript
3. Chen, L., Ali Babar, M., Ali, N.: Variability management in software product lines: a systematic review. In: Proceedings of the 13th International Software Product Line Conference, pp. 81–90 (2009)

4. Czarnecki, K., Antkiewicz, M.: Mapping features to models: a template approach based on superimposed variants. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 422–437. Springer, Heidelberg (2005). https://doi.org/10.1007/11561347_28
5. Fenske, W., Thüm, T., Saake, G.: A taxonomy of software product line reengineering. pp. 1–8. ACM, Sophia Antipolis France (2014)
6. Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M.: Kulesza: evolving software product lines with aspects: an empirical study on design stability. In: Proceedings of 30th International Conference on Software Engineering, ICSE'08. ACM (2008)
7. Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness (2001)
8. Graves, T., Karr, A., Marron, J., Siy, H.: Predicting fault incidence using software change history. IEEE Trans. Software Eng. **26**(7), 653–661 (2000)
9. Heidenreich, F., Kopcsek, J., Wende, C.: FeatureMapper: mapping features to models. In: In: Proceedings of 30th International Conference on Software Engineering. ICSE 2008, Leipzig, Germany, pp. 943–944 (2008)
10. Hinterreiter, D., Grünbacher, P., Prähofer, H.: Visualizing feature-level evolution in product lines: a research preview. In: Madhavji, N., Pasquale, L., Ferrari, A., Gnesi, S. (eds.) REFSQ 2020. LNCS, vol. 12045, pp. 300–306. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-44429-7_21
11. Kasto, N., Whalley, J.: Measuring the difficulty of code comprehension tasks using software metrics **136** (2013)
12. Khan, A.A., Mahmood, A., Amralla, S.M., Mirza, T.H.: Comparison of software complexity metrics. Int. J. Comput. **04**, 19–26 (2016)
13. Khan, F., Musa, S., Tsaramirsis, G., Buhari, S.: SPL features quantification and selection based on multiple multi-level objectives. Appl. Sci. **9**, 18 (2019)
14. Kulakova, E., Aichhorn, M., Schurz, M., Kronbichler, M., Perner, J.: Processing counterfactual and hypothetical conditionals: an fMRI investigation. Neuroimage **72**, 265–271 (2013)
15. Leahy, M.: TyphonJS-ESComplex (2018). https://www.npmjs.com/package/typhonjs-escomplex
16. Liu, J., Zhang, M., Jou, J., Wu, X., Li, W., Qiu, J.: Neural bases of falsification in conditional proposition testing: evidence from an fMRI study. Int. J. Psychophysiol. **85**(2), 249–256 (2012)
17. Loughran, N., Rashid, A.: Framed aspects: supporting variability and configurability for AOP. In: Bosch, J., Krueger, C. (eds.) ICSR 2004. LNCS, vol. 3107, pp. 127–140. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27799-6_11
18. Loughran, N., Rashid, A., Zhang, W., Jarzabek, S.: Supporting product line evolution with framed aspects, p. 5 (2004)
19. Muriana, B., Paul Onuh, O.: Comparison of software complexity of search algorithm using code based complexity metrics. Int. J. Eng. Appl. Sci. Technol. **6**(5), 24–29 (2021)
20. Peitek, N., Apel, S., Parnin, C., Brechmann, A., Siegmund, J.: Program comprehension and code complexity metrics: An fMRI study. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering. ICSE 2021, Madrid, ES, pp. 524–536. IEEE (2021)
21. Perdek, J., Vranić, V.: Lightweight aspect-oriented software product lines with automated product derivation. In: Abelló, A., et al. (eds.) ADBIS 2023. CCIS, vol. 1850, pp. 499–510. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-42941-5_43

22. Perdek, J., Vranić, V.: Matrix based approach for structural and semantic analysis supporting software product line evolution. In: Proceedings of the Tenth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, SQAMIA 2023. CEUR Workshop Proceedings, Bratislava (2023)
23. pure::systems: PLE & code-managing variability in source code (2020). https://youtu.be/RlUYjWhJFkM
24. Rashid, A., Royer, J.C., Rummler, A. (eds.): Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way (2011)
25. Schuster, S., Hawelka, S., Himmelstoss, N.A., Richlan, F., Hutzler, F.: The neural correlates of word position and lexical predictability during sentence reading: evidence from fixation-related fMRI. Lang. Cogn. Neurosci. **35**(5), 613–624 (2020)
26. Sehgal, R., Mehrotra, D.: Predicting faults before testing phase using Halstead's metrics. Int. J. Software Eng. Appl. **9**, 135–142 (2015)
27. Varshosaz, M., Al-Hajjaji, M., Thüm, T., Runge, T., Mousavi, M.R., Schaefer, I.: A classification of product sampling for software product lines. In: Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1, New York, NY, USA, pp. 1–13. SPLC '18, Association for Computing Machinery (2018)
28. Woods, J.: Possible ES6 extensions (2021),.https://github.com/tc39/proposal-decorators/blob/master/EXTENSIONS.md
29. Zuse, H.: Resolving the mysteries of the halstead measures (2005)

Design Pattern Representation and Detection Based on Heterogeneous Information Network

Tao Lu, Xiaomeng Wang(✉), and Tao Jia

College of Computer and Information Science, Southwest University, Chongqing, China
lt1456@email.swu.edu.cn, {wxm1706,tjia}@swu.edu.cn

Abstract. Design patterns (DPs) represent an abstract design approach and are commonly reflected in software code. Design pattern detection (DPD) can help programmers quickly grasp the main structure of the code, thereby reducing the difficulty of software understanding. Micro-structures (MSs), are basic components of the code, have smaller granularity and clearer semantics, and can be used for the DPD task. This paper proposes a syntax-independent code design semantic representation method based on the heterogeneous information network (HIN). A representation and classification of micro-structures is given based on meta-paths of HIN, based on which a DP can be decomposed into MSs. The DPD procedure is then implemented through three steps of DP decomposition, MS matching and MS combination. The method is evaluated on three open-source systems with four other detection tools. The results show that our approach effectively detects DP instances and alleviates the variant problem, which also shows the great potential of applying network science methods to software engineering.

Keywords: design pattern detection · heterogeneous information network · micro-structures · meta-path

1 Introduction

Typically, software developers need to read a lot of software code, whether it is to learn a technology or to maintain a project. The complexity, irregularity and lack of comments [1] of the code bring great challenges to code understanding. Therefore, an abstract perspective on understanding complex code architectures is needed by showing how different parts of the code work in concert to accomplish system goals, which can be achieved through detecting instances of design patterns (DPs).

The concept of DPs was first formally introduced by architect Christopher Alexander [2] and was famous in software engineering thanks to the book *Design Patterns: Elements of Reusable Object-Oriented Software*, authored by Gamma et al. [3]. The book summarizes 23 common DPs known as Gang of Four (GOF)

A. Achilleos et al. (Eds.): ICSR 2024, LNCS 14614, pp. 72–91, 2024.
https://doi.org/10.1007/978-3-031-66459-5_5

DPs. Each DP describes a solution to a general problem that arises during software development, improving code reusability, maintainability, readability, robustness, and security. These solutions have been tested for a long time and have proven to be best practices. The problem now is how to quickly extract and present code DPs in front of developers. Automated design pattern detection (DPD) technology provides the answer and has received continued attention from researchers [4,5].

However, DPD is not a straightforward task. On the one hand, it is a cognitive process for humans to understand software [6], but the knowledge must be formally defined and quantified for machines. Due to informal definitions, GOF DPs cannot be directly identified from the code. On the other hand, some researchers and developers mistakenly consider that GOF DPs are immutable [7,8], while neglecting their flexibility, resulting in a loss of recall for DP variants.

A certain number of DPD methods have been proposed, most of which involve two steps: representation and detection. They differ significantly in terms of the input data format, the form of intermediate data representation, and results. A detailed review can be found in [9]. In addition to the performance of the results, we focus on the generality of the method, i.e., whether the granularity of the representation elements is sufficient to reflect the design semantics, and whether the method is applicable to identify various DP instances (variants included).

To achieve this goal, we refer to micro-structures (MSs) and propose a novel detection method based on the heterogeneous information network (HIN) [10]. In the beginning, we construct a HIN by extracting language-independent design semantic information from the code. Second, we reclassify micro-structures from different catalogs ordered by their structural complexity, which is measured by the length of meta-paths. The GOF DPs can then be decomposed into and represented by a set of basic structures. Finally, instances of these MSs in the HIN are retrieved and combined into DP instances. The results show that the proposed method effectively detects DP instances and their variants. Specifically, the contributions of this work are:

- A syntax-independent and HIN-based representation of object-oriented software structures that expresses design semantics at an appropriate granularity.
- A meta-path based complexity measure for reclassifying and representing MSs.
- A MS-based method for decomposing and representing DPs.
- An efficient DPD method through combining MSs, and an improved strategy based on fuzzy queries over graph databases to alleviate the variant problem.

The structure of this paper is as follows: Sect. 2 presents related work on MSs and our motivation for research. In Sect. 3, our approach is elaborated. Section 4 provides the evaluation results. Finally, conclusions and future work are presented in Sect. 5.

2 Background

Many methods have been proposed for the DPD task. We categorize these methods into the following groups:

- Graph-based methods [11,12]. Such methods typically transform both the system and the DP into a graph structure, and then detect DP instances by matching the approximate graph of the DP via subgraph matching, similarity scoring, and other methods [13].
- Reasoning-based methods [14–16]. The main idea is to convert the matching conditions of DPs into a set of constraint rules, and then analyze the system through a reasoning engine to obtain the matching results of these rules.
- Metric-based methods [17,18]. These methods mainly use computable quantities to study various indicators of software, such as inheritance, number of methods, number of fields, and modifiers. These metrics are then compared with the same category metrics for specific DPs to achieve the detection goal.
- Learning-based methods [19–23]. Learning-based methods select the relevant features of DPs (such as metrics, structural or behavioral properties) and then use the labeled data as training datasets to learn multiple classifiers for different DPs.

In addition to the aforementioned approaches, there are also decomposition-based approaches. At the micro-scale, DPs can be decomposed into smaller components called micro-patterns [24,25], micro-structures [26], etc. A micro-structure (MS) is used to represent a basic component of a DP in this paper. Various MS categories have been proposed for different purposes. Elemental design patterns (EDPs) [27] focus more on the representation of object behavior. Micro patterns proposed by Gil et al. [24] describe various degradation states and behaviors at the class level. Arcelli et al. [28] summarized 46 DP clues according to DP definitions. Most of these catalogs have been proven to be useful [26]. Additionally, they have been integrated into practical inspection tools such as SPQR [29] and Marple [30] to perform the DPD task.

Compared with DPs, these MSs containing specific entities and relations [31], are also components derived from software engineering experience and have clearer semantics so that they can be accurately identified from the code. Therefore, MS-based DPD methods, which use MSs as reasoning metrics [28,29] or use MSs as features for machine learning [22,31,32], typically have higher accuracy and clearer interpretability than other DPD methods.

However, MSs are quite different from each other in scales, e.g., one of the simplest MSs, *Final class* in DP clues [28] means one single class with restricted attributes while one of the most complex MSs, *Retrieve* in EDP [27], involves interactions between multiple classes and methods. Besides, MSs from different catalogs have homonyms, antonyms, or other ambiguities. So it is of interest to integrate existing catalogs, which make the MS-based reusable component representation system more complete.

Thaller [32] integrated and eliminated the ambiguities of the above three catalogs to form a catalog containing 67 MSs. For each MS, a logical formal

language description is provided. These MSs are then used as features for deep learning. Similar efforts to integrate MSs were made by Arcelli et al. [26], on top of three catalogs, they introduced a UML extraction tool called FUJABA [33] and integrated a catalog containing 85 MSs.

Despite the above work, more needs to be done. These integration efforts have merely resolved ambiguities and organized the names and definitions of existing MSs, without considering the expandability of the final catalogy or providing methods for measuring the scales of different MSs. To uniformly organize and analyze these MSs of different sizes, it is necessary to design a new MS stratification method.

In fact, a MS is a kind of network motif [34]. Motifs are defined as building blocks of complex networks. It has been widely applied in biological networks to identify key control mechanisms, e.g., positive and negative feedback motifs. A MS can also be regarded as a motif in the code network, representing a frequently occurring entity connection pattern. Inspired by this, we focus our efforts on proposing a more uniform and extensible MS-catalog similar to the periodic table, such that a DP, or even an entire system, can be fully decomposed into or combined by multiple MSs. Then the DPD task can be easily accomplished. In contrast to other DPD methods, this approach better reflects the essence of DPs, namely reusability. To be clear, this is a shift from studying what a DP contains to what a DP is made of.

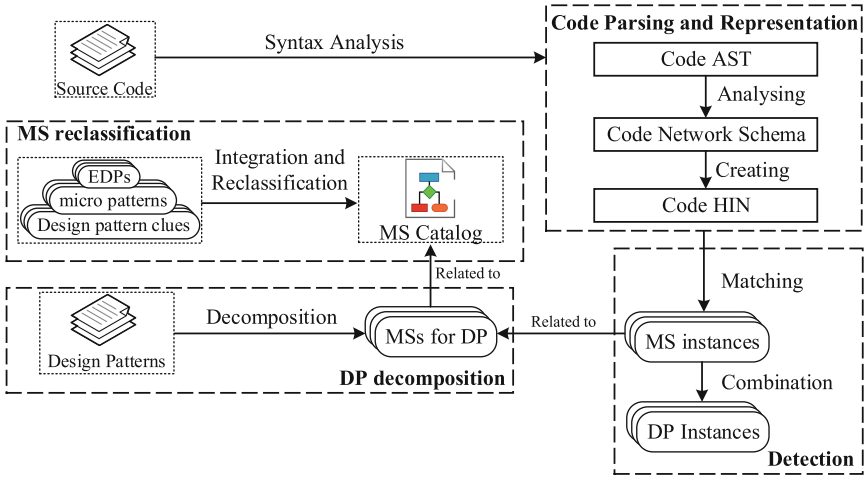

Fig. 1. Design pattern detection process.

3 Approach

As shown in Fig. 1, the method includes four aspects: code parsing and representation, MS reclassification, DP decomposition, and detection. First of all, a code network schema is defined based on the core entities and relations involved

in the object-oriented programming paradigm and then the code is transformed into a heterogeneous information network (HIN) [10,35–37] that conforms to the network schema through abstract syntax tree (AST) parsing. Secondly, a systematic classification method is proposed to construct an extensible MS-catalog hierarchically by structural complexity and three existing catalogs [24,27,28] are integrated. DPs are decomposed and represented on the basis of the proposed MS-catalog. Finally, DP instances are detected by MS matching and combination.

3.1 Code Parsing and Representation

AST is a common structured representation of programs and is widely used in code analysis. However, DPs are independent of specific programming syntax and focus on object states, behaviors, and relations between them. Redundant syntactic details in the AST such as operators and literals, may make data structures more complex and affect the analysis efficiency for the DPD task. Thus, a data representation with object-oriented core elements is required. Like complex systems such as social networks and communication networks, a network can naturally represent the code structure of software. In a code network, nodes represent code entities such as types and fields, and edges represent relations such as inheritance, calls, etc. Since DPs involve multiple types of entities and relations, a HIN is employed to represent the code structure for the DPD task.

Based on the investigation of object-oriented programming languages such as Java and C++, some core entities and relations are selected to form a network schema as shown in Fig. 2 for HIN construction. Types, variables, and executables are the basic entities of the code structure. Due to language differences, types are not specifically distinguished into classes and interfaces. This does not affect the subsequent analysis, because the concept of an interface can generally be embodied by abstract methods. This also avoids some syntactic interference, such as default methods (since JDK 1.8) and marker interfaces in Java. Variables mainly consider fields, parameters, and local variables. In general, each variable is associated with a type. If the variable is a container, it will be associated with both the container type and the element type. Executables are divided into constructors and methods. In addition to the above entities, invocation entities are also considered, including constructor calls and method calls. Invocations can be viewed as messages passing between objects, which facilitates the representation of behavioral semantics between objects.

After the entities are identified, all the potential relations between the entities are listed. In addition to inheritance, inclusion is the most common entity relation, which is marked with "has" in Fig. 2. For example, a type declaration contains entities such as fields, constructors, and methods. As another example, a method or constructor contains entities such as parameters, local variables, and invocations. Some other relations are refinements of associative relations. Each method and variable will correspond to a type, with the former implying the declared return type or the actual return type of a method, and the latter indicating to which type the variable belongs. If necessary, a variable may raise

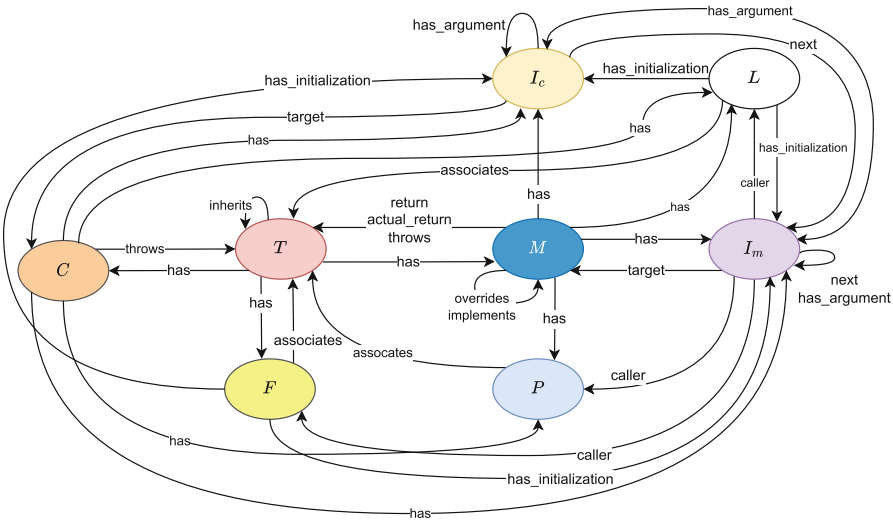

Fig. 2. The network schema for the code HIN. T: type; M: method; C: constructor; I_c: constructor call; I_m: method call; F: field; P: parameter; L: local variable.

an invocation and take the result as its initialization value. All other remaining relations revolve around invocations. On the one hand, an invocation has its caller and its target. The caller is an object, for clarity, a variable associated with a non-primitive type, and the target is usually a method or constructor, which determines whether the invocation is a method call or constructor call. On the other hand, an invocation is self-recursive and can be scaled in two dimensions with other invocations. In the first dimension, there may exist multiple invocations in the same line of code, which forms an invocation chain, and the "next" relation is used to connect them. In the second dimension, an invocation may work as an anonymous argument of another invocation, and this relation is represented by "has_argument".

As a result, we succeeded in setting the granularity of behavioral information at the invocation level. In combination with structural information, we obtain a set of entities and relations, capable of representing any part of the code structure. Once the network schema is defined, the code HIN can be constructed with Algorithm 1.

As a concrete example, the Java program shown in Listing 1 depicts the redirecting behavior of a method. Accordingly, a HIN like Fig. 3 (a) can be obtained analytically. In (a), Type "Bar" redirects its method "operation" to the same method of type "Foo" through a method call. Moreover, there is another invocation in the invocation chain of this method, which is a constructor call that creates an anonymous object by calling the constructor of type "Foo". (b) and (c) describe the class-level and method-level graphs, respectively.

Algorithm 1: The code HIN constructing

Input: *AST*: the code AST; *S*: the network schema;
Output: *HG*: the code HIN;

Function `createHIN`(*AST, S*)**:**
  Initialize *HG* into an empty graph;
  for *each relation paradigm* $R \in S$ **do**
    while *relation r related to R exists in AST* **do**
      create empty source node N_s;
      create empty target node N_s;
      N_s = `createOrFindNode`(*HG,r.source*);
      N_t = `createOrFindNode`(*HG,r.target*);
      create edge between N_s and N_t in *HG*;
      exclude r in AST;
  return *HG*

Function `createOrFindNode`(*HG, Entity*)**:**
  for *each node* $N \in HG$ **do**
    if *N corresponds to Entity* **then**
      return *N*
    create node N^* related to *Entity*;
    return $N*$

```
public class Bar {
    public void operation() {
        new Foo().operation();
        }
}
```

```
public class Foo {
    public void operation() {
    }
}
```

Listing 1: Example code of method redirecting.

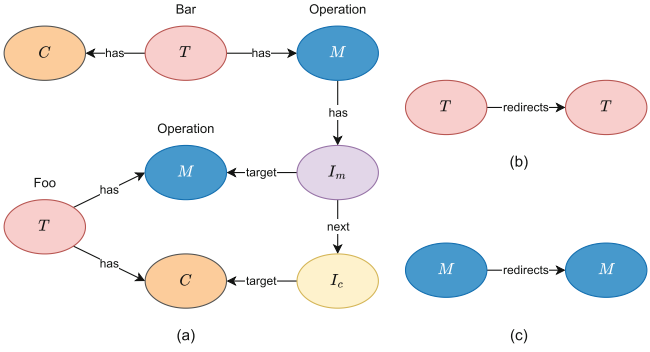

Fig. 3. Examples of method redirecting in different code networks.

It can be seen that the proposed HIN-based representation expresses more semantic information than that based on a class-level or method-level graph.

And, in our approach, the extracted information is both simple and useful: they are combinations of simple binary relations that expose sufficient structural details at an appropriate level of granularity.

3.2 Micro-structure Reclassification

Based on the proposed program structure representation, we propose a scalable multi-level classification strategy for MSs, based on which DPs can be easily decomposed. The definition of MSs is given in Definition 1.

Definition 1. ***Micro-structure (MS).*** *Given a code network HG and its network schema $S = (\mathcal{A}, \mathcal{R})$, where $\mathcal{A}$ and $\mathcal{R}$ are sets of node types and relation types of HG, respectively. One micro-structure $MS = (\mathcal{V}, \mathcal{E})$ is a subgraph of HG. For each node $v \in \mathcal{V}$ and relation $e \in \mathcal{E}$, $type(v) \in \mathcal{A}$ and $type(e) \in \mathcal{R}$, where $type(v)$ and $type(e)$ means the type of a node or an edge, respectively.*

MSs vary in size and complexity. As shown in Eq. 1, a complex MS can be seen as a combination of simpler MSs.

$$MS_k = MS_1 \cup MS_2 \cup \ldots, \tag{1}$$

where "$\cup$" is a union operator which means two graphs are combined by nodes of the same kind. Note that we do not command that every two MS have a common node, as long as some other MS in the graph share a common node with the current MS, which proves that the current MS is part of the whole. When there are similar terms on the right-hand side, Eq. (1) can be further simplified. Specifically, for $MS_1 = (\mathcal{V}_1, \mathcal{E}_1)$ and $MS_2 = (\mathcal{V}_2, \mathcal{E}_2)$, if $type(MS_1) = type(MS_2)$, then

$$MS_1 \cup MS_2 = \begin{cases} {MS_1}^2 = {MS_2}^2, & \mathcal{V}_1 \cap \mathcal{V}_2 \neq \varnothing \\ MS_1 \times 2 = MS_2 \times 2, & \mathcal{V}_1 \cap \mathcal{V}_2 = \varnothing. \end{cases} \tag{2}$$

The reduction rule describes how the combination of MSs of the same type affects the complexity of the composite structure. Specifically, the composition of independent MSs of the same type has no effect on the complexity of the composite structure. In turn, if these same components have common nodes, the coupling of the composite structure increases. This rule can be further applied to the case of multiple MSs.

However, the components of a MS may not be identical to each other, and the decomposition and combination process may be haphazard. In order to distinguish MSs and organize them in a relatively balanced way, a uniform set of notations must be introduced, and meta-paths [38] are adopted here. The meta-path based MS representation is defined as follows.

Definition 2. ***Meta-path based MS representation.*** *A MS is represented as a connection of meta-paths: $MS = \mathcal{P}_1 \cup \mathcal{P}_2 \ldots \cup \mathcal{P}_k$. A meta-path $\mathcal{P}$ is a path defined on a schema $S = (\mathcal{A}, \mathcal{R})$, and is denoted in the form of*

$A_1 \xrightarrow{R_1} A_2 \xrightarrow{R_2} \cdots \xrightarrow{R_l} A_{l+1}$, *which defines multiple relations* $R_1, R_2, \cdots, R_l$ *between entities* $A_1, A_2, \cdots, A_{l+1}$. *Since the directed relation between two entities is usually unique, the meta-path can be simplified to* $\mathcal{P} = A_1 A_2 ... A_{l+1}$.

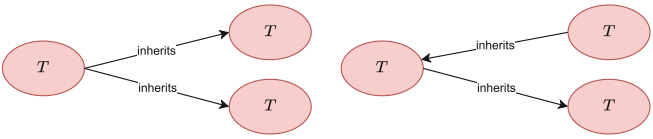

Fig. 4. The TT^2 micro-structure.

Take TT^2 MS shown in Fig. 4 as an example, which is composed of two TT MSs via Eq. (2). TT^2 means there are two "inherits" relations between three "T" nodes. The left TT^2 also appears as "*Joiner*" from micro patterns [24] and "*Interface and class inherited*" from DP clues [28]. The one on the right is an isomer of the one on the left, and since they have the same number of nodes and edges, implying the same complexity, we do not distinguish them in the classification.

MS are reclassified according to their complexity, which is also called the order. The order of a MS is calculated by its length of the critical meta-path as shown in Eq. 3.

$$\mathcal{O}(MS) = \begin{cases} L(\mathcal{P}_{max}(\mathcal{C}(MS))), & N(\mathcal{P}_{max}(\mathcal{C}(MS))) = 1 \\ L(\mathcal{P}_{max}(\mathcal{C}(MS))) + 1, & N(\mathcal{P}_{max}(\mathcal{C}(MS))) > 1, \end{cases} \tag{3}$$

where $\mathcal{C}(MS)$ represents the set of meta-paths in MS that meet condition $\mathcal{C}$; $\mathcal{P}_{max}(X)$ is the set of meta-paths with the longest length in X; $L(Y)$ represents the meta-path length in Y and $N(Z)$ gets the number of elements in the set Z. The filtering condition $\mathcal{C}$ of the meta-path is mainly based on two factors. First, the chosen meta-paths should start from a "T" node because types are the basic entities of the program structure so that any relation can be traced back to a type. Second, in order to ensure the generality of the decomposed MSs, i.e., we expect that the same MSs can be decomposed from different program fragments, and hence the meta-path describing a single characteristic is chosen. If the meta-path is not unique, $L + 1$ would be taken. In fact, the final chosen meta-path is one of the local maxima. We will illustrate this procedure with a concrete example.

In the case of the MS in Fig. 5, the local maximum meta paths that are found are TMI_mF and TMI_mFI_cC. The former describes the behavior of the method call I_m initiated by the field F, and the latter additionally describes the initialization of F by invoking a constructor call I_c, whose target is the constructor C. We choose length 4 for the former as the complexity measure for this MS. The second characteristic described by the latter should be in the

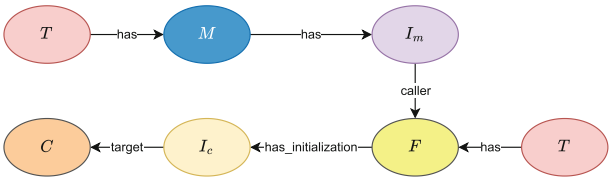

Fig. 5. The $TMI_mF \cup TFI_cC$ micro-structure.

charge of another meta-path TFI_cC. In this relatively balanced way, only meta-paths of length at most 4 are used to describe all the characteristics of the MS. Thus, this MS can be represented as $TMI_mF \cup TFI_cC$ and its order is 5 because it has two meta-paths of length 4 with a common node. Similarly, for the MS TT^2, its order is 3.

In particular, the 1- and 2-order MSs obtained by this classification strategy correspond to individual entities and fundamental binary relations, respectively. From a decomposition point of view, different MSs all make sense. We focus on their structures and present compositional paradigms for them. We have integrated all three MS-catalogs (EDPs, DP clues, and micro patterns) and classified them by their complexity into a new catalog. Thanks to the composition and decomposition rules, the new catalog is extensible to MSs of arbitrary size.

3.3 Design Pattern Decomposition

Based on the previously defined MS-catalog and representation, a DP can be viewed as a combination of multiple small components. The specific form is shown in Definition 3.

Definition 3. ***MS-based design pattern representation.*** *Given a set of DPs $\mathcal{DP}$, for each $DP_i \in \mathcal{DP}$, it can be viewed as a combination of MSs and be decomposed into multiple MSs with*

$$DP_i = MS_1 \cup MS_2 \cup \cdots \cup MS_k \tag{4}$$

This paper mainly considers GOF DPs [3]. Through the analysis, we retain only the key entities and relations of each DP as Xiong et al. [16] did and decompose them into MS-based representations. To explain how to decompose DPs, we illustrate the procedure with the *Singleton* DP and the *Proxy* DP as examples.

Example 1. **Singleton.** *Singleton* is a creational DP that restricts object creation for a class to only one instance. Its DP graph is shown in Fig. 6. *Singleton* requires the constructor to be private, which many DPD tools ignore. *Singleton* can be decomposed into two 3-order MSs and a 2-order MS as shown in Fig. 7.

$$Singleton = TFT \cup TMT \cup TC \tag{5}$$

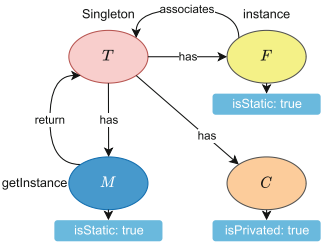

Fig. 6. The *Singleton* design pattern.

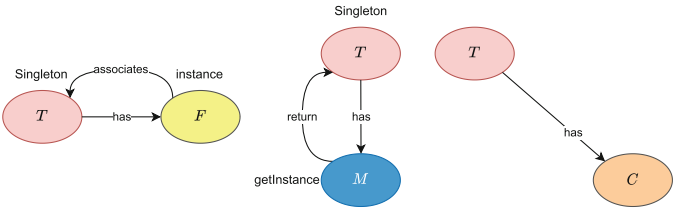

Fig. 7. Decomposing the *Singleton* design pattern.

Example 2. **Proxy.** *Proxy* is a structural DP that provides a placeholder for another object to control access, reduce cost, and reduce complexity. Its DP graph is shown in Fig. 8. *Proxy* can be decomposed into a 5-order MS, a 4-order MS, and a 2-order MS as shown in Fig. 9.

$$\begin{aligned} Proxy =& (TMI_mF \cup TMI_mM) \cup \\ & (TMM \cup TM \cup TT)^2 \cup FT \end{aligned} \tag{6}$$

We do not explicitly declare the "has_field" relation between type "Proxy" and field "realSubject", because this field may be derived from the parent of "Proxy", and the "caller" relation has implicitly limited the origin of this field.

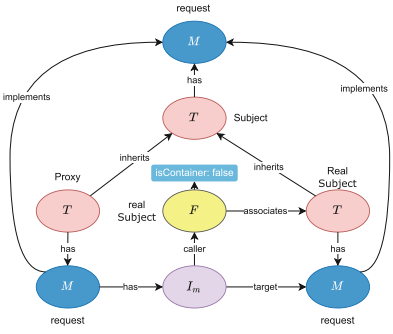

Fig. 8. The *Proxy* design pattern.

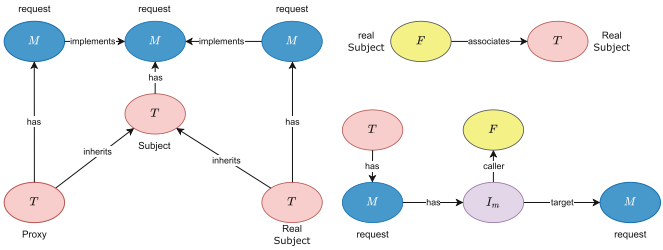

Fig. 9. Decomposing the *Proxy* design pattern.

Based on the proposed MS classification catalog, we have decomposed 21 GOF DPs. Due to the particularity of the *Interpreter* DP and the generality of the *facade* DP, we did not take them into consideration. A full catalog of MSs and DPs is available online [39].

3.4 Design Pattern Detection

Now that the decomposition formulas for DPs are established, we can split the detection task into two steps. First, we find instances of MSs related to different DPs, and then we combine them into DP instances via common nodes, the matching procedure is described by Algorithm 2.

Algorithm 2: Design pattern instance matching

Input: HG: the code HIN; DP: the target design pattern; Set_{MS}: micro-structures decomposed from DP;
Output: Set_{dp}: DP instance set;

Function `matchMSInstance`(HG, Set_{MS})**:**
  initialize Map^{ms}_{MS} from MSs to MS instances;
  for *each* $MS \in Set_{MS}$ **do**
    while $\exists ms \in HG$, *ms is an instance of* MS **do**
      add ms to $Map^{ms}_{MS}[MS]$;
      exclude ms in HG;
  return Map^{ms}_{MS}

Function `matchDPInstance`**:**
  initialize the DP instance set Set_{dp};
  initialize the Cartesian product T;
  Map^{ms}_{MS} = `matchMSInstance`(HG, Set_{MS});
  for *each* $MS \in Set_{MS}$ **do**
    $T = T \quad Map^{ms}_{MS}[MS]$;
  for *each tuple* $\in T$ **do**
    if $\forall ms_i \in tuple, \exists ms_j \in tuple, i \neq j$ *and* $ms_i \cap ms_j \neq \varnothing$ **then**
      add *tuple* into Set_{dp};
  return Set_{dp}

A DP consists of multiple MSs and a DP instance can be obtained from a combination of MS instances belonging to different types of MSs. We compute the Cartesian product of MS instances and consider it as the basis candidate set, wherein for each candidate tuple, only one instance per MS is chosen. Moreover, an MS instance should share at least one common node with other MS instances in the same candidate tuple, otherwise, it is an isolated structure and cannot function as part of a DP. Once the instance sets according to different DPs are obtained, we can get role mappings accordingly, which is the general format for DPD results.

The code HIN is constructed from source code and stored in Neo4j [40], and the matching rules were coded by Cypher [41], which is a declarative graph query language developed by Neo4j as the SQL (Structured Query Language) equivalent language for graph databases. In this way, DPD is transformed into a graph matching task, which is NP-hard. Thanks to previous decomposition work on DPs, we do not have to retrieve the entire graph, but only a set of MSs induced by specific node types and relations, which is much faster. On the one hand, when there is a lack of instances of a certain MS for a DP, we can rule out some candidates in advance to simplify subsequent calculations. On the other hand, intermediate results can be cached. Thus, the query performance of the whole procedure is acceptable.

Based on the above work, we have developed a detection tool (DPD) and found some instances of DPs, but there are many variants that have yet to be discovered. Since the biggest obstacle to DPD is the variety of variants, we analyze the variants and classify them into three types as follows:

- **Overlapping roles**: A class may fulfill the function of multiple roles in the same DP.
- **Scalable inheritance hierarchy**: There may be multiple inheritance levels between roles.
- **Scalable method invocation**: An invocation that describes the key behavior of a DP may not appear directly in a method, but rather in an invocation chain or as an argument. Such an invocation is typically wrapped in an output function or a logging function.

Inspired by fuzzy matching in SQL, we apply fuzzy queries to the graph database to alleviate the variant problem and achieve a 44.8% average improvement in recall. This part of the work is similar to Wegrzynowicz et al. [42]. Take a *FactoryMethod* DP instance from JHotDraw v5.1 for example as shown in Fig. 10. The superclass "*CH.ifa.draw.standard.CompositeFigure*" inherits the grandfather class "*CH.ifa.draw.standard.Abstract-Figure*" without rewriting the method "*decompose*". The subclass "*CH.ifa.draw.figures.GroupFigure*" inherits the parent class and rewrites the grandfather class method. In this way, the instance spans two inheritance levels. We found this instance just by changing the cypher statement from "[:inherits]" and "[:overrides]" to "[:inherits*..]" and "[:overrides*..]", it relaxes the relation restriction from a single fixed restriction to support inheritance hierarchies at zero and above.

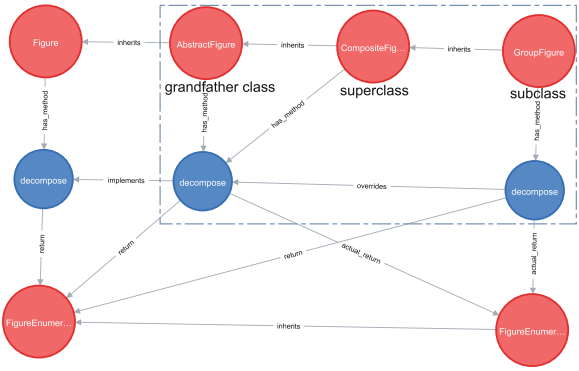

Fig. 10. A *FactoryMethod* design pattern instance.

Fuzzy querying is one of the common ideas for DPD. However, previous work is mainly based on the class-level or method-level graph, whose granularity is too coarse, making it difficult to determine the degree of generalization of query conditions. The granularity of HIN is fine enough to customize various detailed constraints on nodes and relations while ensuring query accuracy. We have written fuzzy query statements for 21 GOF DPs [3] (excluding *Interpreter* and *Facade*) via Cypher. These queries are independent of each other and are able to work alone. Furthermore, based on the proposed MS classification strategy, the user can easily decompose and retrieve any desired DP, such as anti-patterns.

4 Evaluation

In this section, we evaluate the effectiveness of our approach (HIN-DPD) on three open-source Java systems: JHotDraw v5.1 (JHD), JUnit v3.7 (JUN), and QuickUML 2001 (QUM), they are chosen because they contain sufficient DP instances and are used in many other DPD methods. Their sizes are shown in Table 1. We use SPOON [43], a powerful code analysis tool, to build the code AST in the parsing stage.

Table 1. Sizes of three open-source Java systems

Name	Files	Lines	Classes	Methods
JHD	144	8419	173	1332
JUN	78	4886	157	714
QUM	156	9249	228	1096

By analyzing the ASTs of the above three systems, we construct their HINs separately. These HINs are stored in Neo4j, a graph database based on attributed

graphs. We then retrieve DPs in the graph database using the defined fuzzy query statement and obtain excellent experimental results: we not only find more instances of variants but also many DPs in which JDK classes participate. The prototype of the detection tool and detailed results can also be found in the online repository [39].

However, researchers differ in the way instances are counted and the format of the results, which hinders the evaluation work. For example, for the *FactoryMethod* DP instance in Fig. 10, the count may vary from 1 to 6, which increases the difficulty of uniform evaluation. Although an evaluation platform DPB [8] and an evaluation format DPDX [44] have been proposed, they have yet to catch on. In order to evaluate our results, we refer to P-MARt [45] and the dataset offered by Xiong et al. [16] because specific detection instances and role mappings are provided. Another popular dataset, DPB [8], is not available from the official website. We integrate the above two datasets together with the positive instances manually filtered from our results as a benchmark dataset to illustrate the effectiveness of our method. We count the number of instances in the same way as Xiong et al. [16], i.e., a set of key roles of the DP is counted as an instance.

Table 2. Comparison of Ram, DPD, DPF, SparT, P-MARt and HIN-DPD in terms of precision and recall.

Approach	JHD			JUN			QUM			(Aggregated)		
	P[a]	R	F1	P	R	F1	P	R	F1	P	R	F1
RaM	**86.0%**	28.1%	42.4%	56.3%	19.2%	28.6%	**75.0%**	22.3%	34.4%	**72.4%**	23.2%	35.1%
DPD	**95.1%**	25.5%	40.2%	**100.0%**	37.9%	37.9%	/	/	/	**97.6%**	31.7%	39.1%
DPF	**84.3%**	33.3%	47.8%	51.7%	39.5%	39.5%	58.8%	31.9%	41.4%	64.9%	34.9%	42.9%
SparT	**92.5%**	20.3%	33.2%	**75.0%**	38.1%	38.1%	**90.9%**	31.9%	47.2%	**86.2%**	30.1%	39.5%
HIN-DPD	**100.0%**	60.1%	**75.1%**	**100.0%**	59.7%	59.7%	**100.0%**	38.3%	55.4%	**100.0%**	52.7%	63.4%
HIN-DPD*[b]	**100.0%**	**96.7%**	**98.3%**	**100.0%**	**93.2%**	**93.2%**	**100.0%**	**91.5%**	**95.6%**	**100.0%**	**93.8%**	**95.7%**

[a] "P": precision; "R": recall; "F1": F_1-score.

[b] "HIN-DPD*": Improved HIN-DPD after applying fuzzy matching rules.

Table 2 shows the comparison of accuracy, recall, and F1-score achieved by our method with other four methods RaM [46], DPD [47], DPF [48], and SparT [16]. Table 3 shows the comparison of the number of detected instances. Although P-MARt [45] claims to be incomplete and has some false positives, it does provide some correct instances that cannot be identified by existing other DPD methods, some *FactoryMethod* and *Command* instances for example, so we also list it as one of the comparison objects. The results show that our method achieves high accuracy and high recall on all three tested systems. As can be seen from the shared examples in Table 3, our detection results roughly cover the positive instances detected by the other four methods. Compared to HIN-DPD and other methods, HIN-DPD* identifies more variants, most of which are generalizations of the original DPs at the inheritance level and the invocation level. For clarity of comparison with other works, the above comparison does not include those

Table 3. Brief result comparison of Ram, DPD, DPF, SparT, P-MARt and HIN-DPD.

Approach	JHD		JUN		QUM	
	D[a]	TP	D	TP	D	TP
RaM	100	86	16	9	28	20
DPD	82	78	11	11	/	/
DPF	121	102	29	15	51	25
SparT	67	62	16	12	33	30
HIN-DPD	184	184	20	20	36	36
HIN-DPD*	296	296	41	41	86	86
Shared[b]	108		16		34	
(BM)[c]	306		47		94	

[a] "D": Detected instances; "TP": True positives.
[b] Common TP instances.
[c] Benchmark.

Table 4. Extra instances involving JDK classes.

DPs	JHD	JUN	QUM	(Sum)
FactoryMethod	1		8	9
Builder	1		3	4
Prototype	2	2		4
Adapter	21	11	30	62
Bridge	2		3	5
Decorator	1		2	3
Command	19	5	35	59
Observer	5	4	14	23
State/Strategy	1		3	4
Total	53	22	98	173

DP instances involving JDK classes. In fact, we did find some such instances as shown in Table 4.

For the *observer* DP, though we found some instances, we could not locate the "Observer" role, because the "observers" field is a generic container and its item types are not explicitly specified at declaration time. For C++, the true type can be obtained at runtime, while this does not work for Java due to type erasure. This also hinders our retrieval of *Iterator* DPs.

5 Conclusion and Future Work

A code representation based on HIN is proposed for the design elements of OOP. Compared to AST and UML, the proposed HIN-based representation removes

the grammatical factors while retaining the necessary design semantics to represent and detect DPs more accurately and efficiently. In order to give a unified expression for MSs and DPs, a meta-path based method for MS classification and representation is proposed. Based on the above foundation, the DPD task can be easily implemented with three steps decomposition, matching, and combination. The effectiveness of our method is validated by the evaluation of three open-source systems and comparison with four other DPD methods. The proposed MS catalog can also be used in other DPD methods (like reasoning-based methods and learning-based methods) to improve the performance.

The proposed method still needs further improvement and exploration. For the DPD task, continuous improvements can be made in terms of data and algorithms. For example, dataflow graphs could be introduced to track type conversions of generic variables to solve the role mapping deletion problem, and faster graph matching algorithms could be used to improve performance. Moreover, we expect the DPD community to further facilitate the construction and generalization of a unified public evaluation platform with more datasets supporting DPD tasks for different program languages so that empirical research in the DPD field can be further facilitated.

It is exciting to apply network science methods to software engineering. HIN-based representation data can be applied to multiple graph learning tasks such as DP completion and DP recommendation via link prediction, novel DP mining via motif mining techniques, and analyzing software evolution by constructing temporal networks. This paper serves as a foundation for our subsequent research. In the future, the focus will be on evaluating the quality of the proposed network schema and refining its support for other languages in addition to its application to other code analysis tasks.

Acknowledgement. The authors would like to express our great appreciation to the editors and reviewers. This work is supported by National Natural Science Foundation of China (NSFC) (No. 62006198), Chongqing Innovative Research Groups (No. CXQT21005) and the Fundamental Research Funds for the Central Universities (No. SWU-XDJH202303).

References

1. Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D.: The promises and perils of mining github. In: Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 92–101 (2014)
2. Alexander, C.: A Pattern Language: Towns, Buildings, Construction. Oxford University Press, Oxford (1977)
3. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Pearson Deutschland GmbH (1995)
4. Ampatzoglou, A., Charalampidou, S., Stamelos, I.: Research state of the art on GOF design patterns: a mapping study. J. Syst. Softw. **86**(7), 1945–1964 (2013)
5. Mayvan, B.B., Rasoolzadegan, A., Yazdi, Z.G.: The state of the art on design patterns: a systematic mapping of the literature. J. Syst. Softw. **125**, 93–118 (2017)

6. Maalej, W., Tiarks, R., Roehm, T., Koschke, R.: On the comprehension of program comprehension. ACM Trans. Softw. Eng. Methodol. (TOSEM) **23**(4), 1–37 (2014)
7. Lauder, A., Kent, S.: Precise visual specification of design patterns. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 114–134. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054089
8. Fontana, F.A., Caracciolo, A., Zanoni, M.: DPB: a benchmark for design pattern detection tools. In: 2012 16th European Conference on Software Maintenance and Reengineering, pp. 235–244. IEEE (2012)
9. Yarahmadi, H., Hasheminejad, S.M.H.: Design pattern detection approaches: a systematic review of the literature. Artif. Intell. Rev. **53**, 5789–5846 (2020)
10. Sun, Y., Han, J.: Mining heterogeneous information networks: a structural analysis approach. ACM SIGKDD Explor. Newsl. **14**(2), 20–28 (2013)
11. Dong, J., Sun, Y., Zhao, Y.: Design pattern detection by template matching. In: Proceedings of the 2008 ACM Symposium on Applied computing, pp. 765–769 (2008)
12. Pradhan, P., Dwivedi, A.K., Rath, S.K.: Detection of design pattern using graph isomorphism and normalized cross correlation. In: 2015 Eighth International Conference on Contemporary Computing (IC3), pp. 208–213. IEEE (2015)
13. Rao, R.S.: A review on design pattern detection. Int. J. Eng. Res. Technol. **8**(11), 756–762 (2019)
14. Al-Obeidallah, M.G., Petridis, M., Kapetanakis, S.: A structural rule-based approach for design patterns recovery. In: Software Engineering Research, Management and Applications, pp. 107–124 (2018)
15. Kirasić, D., Basch, D.: Ontology-based design pattern recognition. In: Lovrek, I., Howlett, R.J., Jain, L.C. (eds.) KES 2008. LNCS (LNAI), vol. 5177, pp. 384–393. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85563-7_50
16. Xiong, R., Li, B.: Accurate design pattern detection based on idiomatic implementation matching in java language context. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 163–174. IEEE (2019)
17. Dubey, S.K., Sharma, A., Rana, A.: Comparison study and review on object-oriented metrics. Global J. Comp. Sci. Technol. **12**(7), 39–48 (2012)
18. Issaoui, I., Bouassida, N., Ben-Abdallah, H.: Using metric-based filtering to improve design pattern detection approaches. Innov. Syst. Softw. Eng. **11**, 39–53 (2015)
19. Hussain, S., Keung, J., Sohail, M.K., Khan, A.A., Ilahi, M.: Automated framework for classification and selection of software design patterns. Appl. Soft Comput. **75**, 1–20 (2019)
20. Dwivedi, A.K., Tirkey, A., Rath, S.K.: Software design pattern mining using classification-based techniques. Front. Comp. Sci. **12**, 908–922 (2018)
21. Nazar, N., Aleti, A., Zheng, Y.: Feature-based software design pattern detection. J. Syst. Softw. **185**, 111179 (2022)
22. Thaller, H., Linsbauer, L., Egyed, A.: Feature maps: a comprehensible software representation for design pattern detection. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 207–217. IEEE (2019)
23. Ardimento, P., Aversano, L., Bernardi, M.L., Cimitile, M.: Design patterns mining using neural sub-graph matching. In: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, pp. 1545-1553. ACM, New York (2022). https://doi.org/10.1145/3477314.3507073

24. Gil, J., Maman, I.: Micro patterns in java code. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 97–116 (2005)
25. Kim, S., Pan, K., Whitehead Jr, E.J.: Micro pattern evolution. In: Proceedings of the 2006 International Workshop on Mining Software Repositories, pp. 40–46 (2006)
26. Fontana, F.A., Maggioni, S., Raibulet, C.: Understanding the relevance of micro-structures for design patterns detection. J. Syst. Softw. **84**(12), 2334–2347 (2011)
27. Smith, J.M., Stotts, D.: Elemental design patterns: a formal semantics for composition of oo software architecture. In: 27th Annual NASA Goddard/IEEE Software Engineering Workshop, Proceedings, pp. 183–190. IEEE (2002)
28. Fontana, F.A., Zanoni, M., Maggioni, S.: Using design pattern clues to improve the precision of design pattern detection tools. J. Object Technol. **10**(4), 1–31 (2011)
29. Smith, J.M., Stotts, D.: SPQR: flexible automated design pattern extraction from source code. In: 18th IEEE International Conference on Automated Software Engineering, Proceedings, pp. 215–224. IEEE (2003)
30. Fontana, F.A., Zanoni, M.: A tool for design pattern detection and software architecture reconstruction. Inf. Sci. **181**(7), 1306–1324 (2011)
31. Arcelli, F., Cristina, L.: Enhancing software evolution through design pattern detection. In: Third International IEEE Workshop on Software Evolvability 2007, pp. 7–14. IEEE (2007)
32. Thaller, H.: Towards Deep Learning Driven Design Pattern Detection/submitted by Hannes Thaller. Ph.D. thesis, Universität Linz (2016)
33. Nickel, U., Niere, J., Zündorf, A.: The fujaba environment. In: Proceedings of the 22nd International Conference on Software Engineering, pp. 742–745 (2000)
34. Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., Alon, U.: Network motifs: simple building blocks of complex networks. Science **298**(5594), 824–827 (2002). https://doi.org/10.1126/science.298.5594.824
35. Sun, Y., Yu, Y., Han, J.: Ranking-based clustering of heterogeneous information networks with star network schema. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 797–806 (2009)
36. Han, J.: Mining heterogeneous information networks by exploring the power of links. In: Gama, J., Costa, V.S., Jorge, A.M., Brazdil, P.B. (eds.) DS 2009. LNCS (LNAI), vol. 5808, pp. 13–30. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04747-3_2
37. Shi, C., Li, Y., Zhang, J., Sun, Y., Philip, S.Y.: A survey of heterogeneous information network analysis. IEEE Trans. Knowl. Data Eng. **29**(1), 17–37 (2016)
38. Sun, Y., Han, J., Yan, X., Yu, P.S., Wu, T.: Pathsim: meta path-based top-k similarity search in heterogeneous information networks. Proc. VLDB Endow. **4**(11), 992–1003 (2011)
39. Lu, T.: Dataset for: Design pattern representation and detection based on heterogeneous information network. https://github.com/lt3355/Dataset4HIN-DPD-master. Accessed 18 Apr 2024
40. Kemper, C., Kemper, C.: Getting to know neo4j. Beginning Neo4j pp. 13–23 (2015)
41. Francis, N., et al.: Cypher: an evolving query language for property graphs. In: Proceedings of the 2018 International Conference on Management of Data, pp. 1433–1445 (2018)
42. Węgrzynowicz, P., Stencel, K.: Relaxing queries to detect variants of design patterns. In: 2013 Federated Conference on Computer Science and Information Systems, pp. 1571–1578. IEEE (2013)

43. Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L.: Spoon: a library for implementing analyses and transformations of java source code. Softw. Pract. Exp. **46**, 1155–1179 (2015). https://doi.org/10.1002/spe.2346. https://hal.archives-ouvertes.fr/hal-01078532/document
44. Kniesel, G., et al.: Dpdx–towards a common result exchange format for design pattern detection tools. In: 2010 14th European Conference on Software Maintenance and Reengineering, pp. 232–235. IEEE (2010)
45. Guéhéneuc, Y.G.: P-mart: Pattern-like micro architecture repository. In: Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories, pp. 1–3 (2007)
46. Rasool, G., Mäder, P.: A customizable approach to design patterns recognition based on feature types. Arab. J. Sci. Eng. **39**, 8851–8873 (2014)
47. Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.T.: Design pattern detection using similarity scoring. IEEE Trans. Softw. Eng. **32**(11), 896–909 (2006)
48. Bernardi, M.L., Cimitile, M., Di Lucca, G.: Design pattern detection using a DSL-driven graph matching approach. J. Softw. Evol. Process **26**(12), 1233–1266 (2014)

An Ontology-Based Representation for Shaping Product Evolution in Regulated Industries

Barbara Gallina[1(✉)], Henrik Dibowski[2], and Markus Schweizer[2]

[1] Mälardalen University, Västerås, Sweden
barbara.gallina@mdu.se
[2] Robert Bosch GmbH, Renningen, Germany
{henrik.dibowski,markus.schweizer}@de.bosch.com

Abstract. Compliance management is a challenging activity in regulated industries. This is due to the need of navigating rapidly shifting requirements. In the automotive domain products (items) evolve rapidly creating a product line in time in addition to the one in space. The product variability is constrained by legislation, standards, etc. whose applicability may vary depending on the jurisdiction. In this paper, we focus on the legislation and its impact on the product in terms of inclusion/exclusion of product features. We exploit the Semantic Web technologies and we propose an ontology-based representation for managing product variability in compliance with legislation. Specifically, we provide the representation in Resource Description Framework (RDF) and we introduce reusable SHACL (Shapes Constraint Language) constraints to shape the RDF graphs. We illustrate our proposal by considering US and UNECE regulations and their impact on the window lifter.

Keywords: Compliance Management · Product Evolution · Feature Diagrams · Ontologies · Reusable SHACL Constraints · Variability

1 Introduction

Compliance management is a challenging activity in (highly) regulated industries. This is due to the need of handling high volumes of data and navigating rapidly shifting requirements. In the automotive domain, for instance, products (items) evolve rapidly creating a product line in time in addition to the one in space. The product variability is constrained by legislations, standards, guidelines, etc. whose applicability may vary depending on the jurisdiction. Hence, the variability to be managed is not only at the technical product level but it also comprises the management of the variability related to potentially applicable and overlapping legislations, standards, guidelines, etc. This implies that activities focused on modelling these different types of features need to be supported.

Supported by 4DSafeOps #49 project, https://www.software-center.se.

A. Achilleos et al. (Eds.): ICSR 2024, LNCS 14614, pp. 92–102, 2024.
https://doi.org/10.1007/978-3-031-66459-5_6

All these pieces of variable information contribute to building the product's pedigree [2]. In [4], authors have elaborated a layered structure where the interdependencies of different variable artifacts (applicable regulations, applicable standards, etc. and consequent implication on product evolution) are visualized. Via this layered structure, different departments within a highly regulated industry are expected to identify features and characterise what varies and what remains a commonality in their domain of expertise. We recognize that layered structure is actually present within industrial settings. However, based on experience and as emerged in a recently conducted survey [1], we also recognize that to manage the variability at different levels, there is a lack of suitable modeling approaches that enable a more comprehensive and systematic modeling as well as lack of tools allowing for interoperability. Specifically, authors in [1] foresee a new generation of software variability tools with better visualization capabilities where feature models can be shown not as a whole, but separated into functional areas and easily to be managed. In [6], authors elaborate a set of principles for feature modelling, including the principle "split large models and facilitate consistency with interface models", as done in [5]. Interface models are, however, an ad-hoc solution and not a systematic one. To enable different departments to interoperate while managing the variability at their own level (i.e., to split large models while managing the interfaces), we propose an ontology-based representation for managing product variability and re-configuration at different levels. To do that, first, we provide a feature model metamodel, formalized as an ontology, given in RDF, augmented with SHACL constraints to shape/validate the RDF graphs. We also provide reusable SHACL constraints for characterizing the nature of the features (in terms of their variability type). To illustrate our proposal, we focus our attention on two layers, i.e., the legislations and their impact on the product in terms of inclusion/exclusion of product features. Specifically, we consider US and UNECE regulations and their impact on the window lifter. We show how to represent and validate interrelated feature diagrams representing legislations and window lifters. The rest of the paper is organised as follows. In Sect. 2, we recall essential background information. In Sect. 3, we present and illustrate our ontology-based representation. In Sect. 4, we discuss related work. Finally, in Sect. 5, we summarise and sketch future work.

2 Background

Feature diagrams are a graphical representation of a hierarchically arranged set of **features**, i.e., system properties that capture commonalities or discriminate among systems in a family. From a semantics perspective, feature diagrams are trees that are composed of nodes and directed edges. Hence, we sometimes call them feature trees. The tree root represents a feature that is progressively decomposed using mandatory, optional, alternative (exclusive-OR features) and OR-features. In feature diagrams, assuming (during configuration) a top-down selection of features, mandatory features are features that are always included in every product. Features that are not necessarily included in every product

may vary. Variation points are features that have at least one direct variable sub-feature (i.e. as one of its children). The tree can be constrained by formulating cross-cutting constraints (requires/excludes) that impose presence/absence of features based on presence/absence of other features. In this paper, we call feature tree variants the feature tree configurations.

Power-operated window lifters (WLs) regulate the movement of a side car window pane. WLs represent a standard feature mounted on every modern car. A WL is always composed of mechanics, electronics, and the connector to receive the power input. The mechanics is composed of a set of components, including a drive gear (mandatory component), composed by a direct current (DC) motor and toothed and worm gears. The DC motor is responsible for converting the electrical energy into mechanical energy, needed to actuate the movement (lifting or lowering) of the window pane. The electronics is also composed of a set of components, including a microcontroller (MC) unit, which, based on the sensed information and on the received input, controls the speed and torque of the DC motor. The MC is composed of hardware and software, which may vary. Different implementations can be selected depending on the to-be-satisfied needs.

WL-related and self-reversal/anti pinch systems-focused legal requirements applicable in USA and EU. Both, the USA and EU legislations, specify the same maximum pressure of 100N in their standards. Hence, this is a mandatory feature. However, in the USA, as part of the Code of Federal Regulations, specifically the part on Federal Motor Vehicle Safety Standards (FMVSS), Section 571.118 (Standard No. 118; Power-operated window, partition, and roof panel systems), it is stated that to detect a pinch situation, tests must be conducted with a specific test rod stiffness of 65 N/mm (S8.1.(b)). In Europe, according to the UNECE regulation 21 (R21) [7], the required stiffness for the test rods is 10 N/mm (Paragraph 5.8.3.1.3.). These different requirements reveal a variability of type XOR, specifying that UNECE and US legislations are in XOR and that depending on which one is applicable, corresponding requirements (features) have to be satisfied. These different requirements have an implication on the design of the system (i.e., call for different algorithmic solutions at software level), since a stiffer test rod requires faster reaction times as the pressure increases faster.

Resource Description Framework (RDF) [10] allows users to describe resources on the Web in a machine-understandable format. The abstract syntax of RDF has two key data structures: RDF graphs as sets of subject-predicate-object triples, and RDF datasets for organizing several RDF graphs.**Web Ontology Language (OWL)** [8] extends RDFS by adding advanced constructs for defining the semantics of RDF statements. This enables the formulation of constraints related to e.g., cardinality, restrictions of values. The main building blocks of an OWL ontology are classes. A *class* defines a group of individuals that belong together because they share the same properties. Classes can be organised in a specialisation hierarchy using *subClassOf*. **Shapes Constraint Language (SHACL)** [11] is a language for defining constraints, called "SHACL

shapes", against which RDF graphs can be validated. SHACL constraints can be interpreted and processed by a SHACL engine, which checks and validates the SHACL constraints. In case there are facts in an ontology model or knowledge graph (KG) which violate any SHACL constraints, the SHACL engine reports a constraint violation, along with a description. This is then shown to the user. SHACL has its own vocabulary (e.g., a shape is specified with the construct sh:shape), but it uses RDF and RDFS vocabulary to define types, classes, subclasses, properties, lists and resources. **SPARQL** [9] stands for SPARQL Protocol and RDF Query Language. SPARQL can be used to formulate queries across diverse data sources (which can be stored natively as RDF graphs).

3 Ontology-Based Representation

In this section, we present our three contributions. **Contribution-1** consists of a feature tree metamodel, formalised as ontology. The metamodel allows for modeling individual feature trees of any domain and of arbitrary size. The feature tree metamodel is defined as UML (Unified Modeling Language) class diagram, shown in Fig. 1. It comprises two classes and five relationships in total.

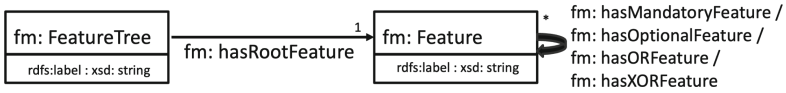

Fig. 1. Feature tree metamodel

fm:FeatureTree is the class that represents feature trees. Each feature tree has exactly one root node, which is defined via the relationship `fm:hasRootFeature`. The feature it refers to is the root feature, i.e. the top feature in the feature tree. Each feature can comprise several sub-features, which can be either mandatory features (relationship `fm:hasMandatoryFeature`), optional features (relationship `fm:hasOptionalFeature`), alternative features (relationship `fm:hasORFeature`) or exclusive features (relationship `fm:hasXORFeature`). With this feature tree metamodel, arbitrary feature trees can be expressed. This metamodel can be easily formalized as OWL ontology, by defining the classes as OWL class, and the relationships as OWL object property. Figure 2 shows an excerpt of the feature tree ontology in RDF syntax.

```
fm:FeatureTree
 a owl:Class ;
 rdfs:label "Feature Tree" ;
.

fm:Feature
 a owl:Class ;
 rdfs:label "Feature" ;
.

fm:hasRootFeature
 a owl:ObjectProperty ;
 rdfs:label "has root feature" ;
 rdfs:domain fm:FeatureTree ;
 rdfs:range fm:Feature ;
.

fm:hasMandatoryFeature
 a owl:ObjectProperty ;
 rdfs:label "has mandatory feature" ;
 rdfs:domain fm:Feature ;
 rdfs:range fm:Feature ;
.
```

Fig. 2. Excerpt of the feature tree ontology in RDF syntax.

The metamodel ontology constitutes a reusable upper ontology, which can be imported and refined by a dedicated feature ontology, specialized for a certain domain and use case. Therein, the specific feature classes and relationships of the domain and use case can be defined. To link it with the metamodel, the feature classes can be defined as subclasses of class `fm:Feature`, and the relationships can be defined as subproperties of the object property `fm:hasMandatoryFeature`, `fm:hasOptionalFeature`, `fm:hasORFeature` or `fm:hasXORFeature`. This is shown for the class `sa:LegalRequirements` and the object property `sa:comprises Jurisdiction` in Fig. 3.

```
sa:LegalRequirements
 a owl:Class ;
 rdfs:label "Legal Requirements" ;
 rdfs:subClassOf fm:Feature ;
 sh:rule
 co:SPARQLRule_AutoCreateMandatoryFeatures ;
 .

sa:comprisesJurisdiction
 a owl:ObjectProperty ;
 rdfs:subPropertyOf fm:hasMandatoryFeature ;
 .
```

Fig. 3. LegalRequirements Class.

A feature ontology representing and organizing the features of the regulations is shown in Fig. 4 as UML class diagram. This diagram translates the information, which was provided in Subsect. 2. The depicted classes represent OWL classes, and the associations between them OWL object properties defined in the feature ontology. The superclass of each class, and the superproperty of each object property from the metamodel are shown as UML stereotypes using the "`<<...>>`" notation. This feature ontology will be used in the following for introducing and demonstrating the reusable SHACL constraints.

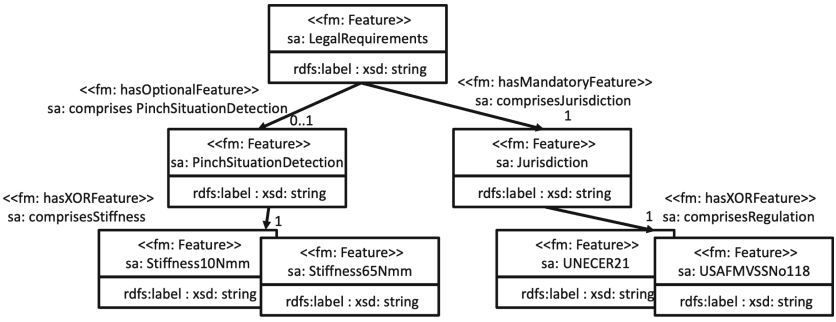

Fig. 4. A feature ontology representing the overlapping legislations.

A feature ontology representing and organizing the features of the set of window lifters is shown in Fig. 5 as UML class diagram.

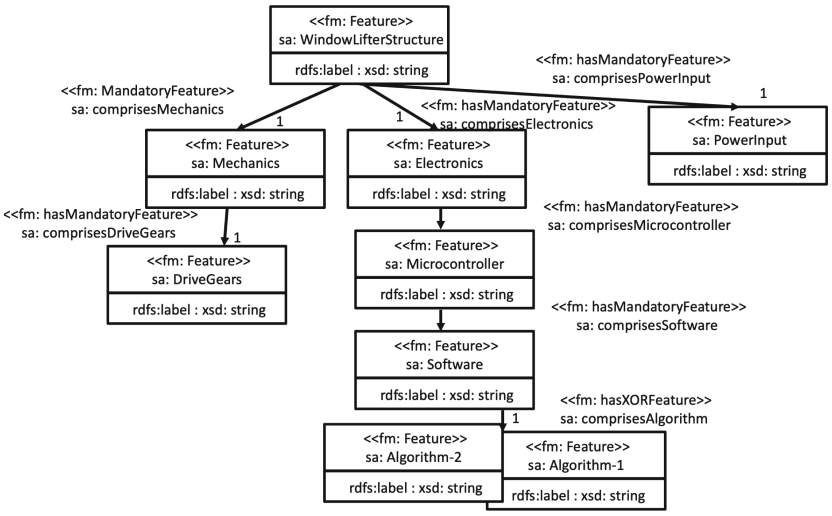

Fig. 5. A feature ontology representing a set of window lifters.

Contribution-2 consists of the validation of Feature Tree Variants with SHACL. Feature tree variants are to be modeled via instances of the specific feature ontology and relationships between them. Instances of a specific `fm:Feature` subclass represent a chosen feature. The relationships defined in the feature ontology are to be used for inter-relating the instances, so that the instance model represents a valid instantiation of the feature tree. The following RDF snippet demonstrates that on the example of a feature instance `sd:LegalRequirements_1` and its two relationships to (sub-)feature instances:

```
sd:LegalRequirements_1
  a sa:LegalRequirements ;
  sa:comprisesJurisdiction sd:Jurisdiction_1 ;
  sa:comprisesPinchSituationDetection sd:PinchSituationDetection_1:
.
```

For validating a feature tree variant, we propose to use SHACL, which means that a feature ontology needs to be augmented with several SHACL statements. To realise that, the OWL classes in an ontology need to be extended to SHACL node shapes, and be related to the SHACL property shapes. This is shown in the following example for the class `sa:LegalRequirements` in RDF:

```
sa:LegalRequirements
  a owl:Class , a sh:NodeShape ;
  rdfs:label "Legal Requirements" ;
  rdfs:subClassOf fm:Feature ;
  sh:property sa:LegalRequirements-comprisesJurisdiction ;
  sh:property sa:LegalRequirements-comprisesMaxSqueezingForce ;
```

```
  sh:property sa:LegalRequirements-comprisesPinchSituationDetection ;
  sh:sparql co:SPARQLConstraint_RequiresDependency ;
.
```

In this example, `sa:LegalRequirements` is declared to be a SHACL node shape via the statement "`sa:LegalRequirements a sh:NodeShape`", besides being a `owl:Class`. Furthermore, via the SHACL property `sh:property`, several SHACL property shapes are attached to it. Each SHACL property shape is specific for one relationship of the class. To a SHACL property shape, certain constraints can be attached, as shown in what follows.

Contribution-3 consists of reusable SHACL constraints for mandatory, optional, OR, XOR, and requires feature relations, as explained in the following using examples, taken from the feature ontology shown in Fig. 4. The constraints are defined on the level of SHACL property shapes with SHACL core constraint components and are specifically designed to support a top-down based creation of feature tree variants. This means that a feature tree configuration process starts with the instantiation of the root feature, followed by the features on the second, third, fourth level and so on. At each (intermediate) step of this top-down process, the SHACL constraints are capable of validating the correctness and completeness of the current feature tree variant.

SHACL Constraint Pattern for Mandatory Features. A mandatory feature can be realized by a SHACL property shape that constraints the cardinality of the object property to have exactly one value, accomplished with the SHACL cardinality constraint components `minCount=1` and `maxCount=1`. This can be seen in the following RDF snippet:

```
sa:LegalRequirements-comprisesJurisdiction
  a sh:PropertyShape ;
  sh:path sa:comprisesJurisdiction ;
  sh:class sa:Jurisdiction ;
  sh:name "comprises jurisdiction" ;
  sh:maxCount 1 ;
  sh:minCount 1 ;
.
```

This SHACL property shape requires each instance of the class `sa:Legal Requirements` to have exactly one `sa:comprisesJurisdiction` relation to an instance of class `sa:Jurisdiction`, which represents the mandatory feature.

SHACL Constraint Pattern for Optional Features. An optional feature can be realized by a SHACL property shape that constraints the cardinality

```
sa:LegalRequirements-comprisesPinchSituationDetection
 a sh:PropertyShape ;
 sh:path sa:comprisesPinchSituationDetection ;
 sh:class sa:PinchSituationDetection ;
 sh:name "comprises pinch situation detection" ;
 sh:maxCount 1 ;
 .
```

Fig. 6. RDF snippet representing the optional feature.

of the object property to zero or one value, accomplished with the SHACL cardinality constraint component `maxCount=1`. The absence of a `sh:minCount cardinality` constraint component makes the relationship optional. This can be seen in Fig. 6.

This SHACL property shape defines that each instance of the class `sa:Legal Requirements` may have one (optional) `sa:comprisesPinchSituation-Detection` relation to an instance of class `sa:PinchSituationDetection`, which represents the optional feature.

SHACL Constraint Pattern for XOR Features. An "exclusive or" feature (XOR) can be realized by a SHACL property shape that constraints the cardinality of the object property to have exactly one value, accomplished with the SHACL cardinality constraint components `minCount=1` and `maxCount=1`, and that defines the XOR features via SHACL `sh:or` as alternative range classes. This SHACL property shape requires each instance of class `sa:PinchSituation Detection` to have exactly one `sa:comprisesStiffness` relation to an instance of either class `sa:Stiffness10N` or `sa:Stiffness65N`, which represent the XOR features. This can be seen in Fig. 7.

```
sa:PinchSituationDetection-comprisesStiffness
  a sh:PropertyShape ;
  sh:path sa:comprisesStiffness ;
  sh:maxCount 1 ;
  sh:minCount 1 ;
  sh:name "comprises stiffness" ;
  sh:nodeKind sh:IRI ;
  sh:or ([sh:class sa:Stiffness10N;]
         [sh:class sa:Stiffness65N;]);
.
```

Fig. 7. RDF snippet representing the XOR feature.

SHACL Constraint Pattern for OR Features. Alternative features (OR) can be realized by a SHACL property shape that constraints the cardinality of the object property to one or more values, accomplished with the SHACL cardinality constraint component `minCount=1`, and that defines the OR features via SHACL `sh:or` as alternative range classes.

SHACL Constraints for Requires. Constraints that express comprehensive interdependencies between different features can be modelled in SHACL as SPARQL-based constraints. We propose to specify a "requires" relationship between one feature of one feature model, and another feature of another feature model as SPARQL-based constraint. This constraint detects requires dependencies between features of different feature models. It requires a relationship of type `fm:requiresFeature` to be defined in the feature ontology between the requiring class (subject) and the required class (object). As an example, we consider that this relationship exists between the feature `sa:USAFMVSSNo118` in Fig. 4 and the feature `sa:Algorithm-1` in Fig. 5. If the feature `sa:USAFMVSSNo118` is selected

by the legal requirements configuration, `sa:Algorithm-1` is required in the window lifter configuration. The constraint detects violations of specific requires dependencies, i.e. two features where one requires the other and that not both appear at the same time in different feature models. The requires constraint is generic and reusable, i.e. it can be associated with any root feature class of a feature model, and does not require adaptations when doing so. The requires constraint, realized as SPARQL-based constraint, is shown in the following:

```
co:SPARQLConstraint_RequiresDependency
  a sh:SPARQLConstraint ;
  sh:message "The feature {?instanceRequiringFeatureClass} of type
  {?requiringFeatureClass} requires the feature {?featureClass}" ;
  sh:select """PREFIX sh: <http://www.w3.org/ns/shacl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT $this ?requiringFeatureClass ?instanceRequiringFeatureClass
    ?featureClass ?instanceFeatureClass
WHERE { $this a ?rootFeatureClass.
  ?rootFeatureClass ((sh:property/sh:class)|
      (sh:property/sh:or/rdf:rest*/rdf:first/sh:class))+
      ?featureClass.
  ?requiringFeatureClass fm:requiresFeature ?featureClass.
  ?instanceRequiringFeatureClass a ?requiringFeatureClass.
  FILTER NOT EXISTS { ?instanceFeatureClass a ?featureClass} }""";
.
```

The functional principle of the constraint is the following: The constraint iterates over all instances of the root feature class it is associated with (variable `$this`). From that feature class it recursively explores the entire feature tree to all contained feature classes. That is realized by a comprehensive recursive SPARQL path (operator "`+`") comprising two alternative subpaths (operator "|"). The first alternative subpath covers mandatory and optional relationships, which are defined in SHACL via `sh:class` property. The second subpath covers OR and XOR relationships, which make use of the SHACL `sh:or`. Here a comprehensive iterating through the RDF list comprising the `sh:or`-related classes is required via `rdf:rest` and `rdf:first` properties. For each feature class of the feature tree found via the SPARQL path, the existence of an incoming `fm:requiresFeature` relation from another feature class is checked. Then, it is checked if the requiring class has an instance, but the required class not. If so, this means that the requires constraint is violated, and the requiring instance, along with its class, is reported back to the user. The SHACL engine therefore constructs (and reports about) a constraint violation message from the string defined in `sh:message`. At runtime the SHACL engine inserts data from the KG via SPARQL variables into the placeholders, e.g. `{?featureClass}`. The respective variable binding when executing the SPARQL-based constraint will be inserted into the message.

4 Related Work

In [12], authors propose an ontology to formally specify feature models. Their ontology, however, is mainly constituted of a set of OWL classes with limited expressiveness. Despite the intention of proposing an OWL representation for the knowledge contained in feature models, representing all possible feature model semantics and despite the intention of formulating SWRL (Semantic Web Rule Language) rules to ensure the consistency of the feature model and detect contradicting or conflicting knowledge in the model, the expressiveness is rather limited and allows limited checks. In addition, rules are not conceived in an explicit reusable manner. In [3], the author proposes a mapping between feature trees (given in a domain-specific language) and RDF graphs enriched with SHACL constraints. The SHACL constraints formulated in his work are defined specifically for instances of features, i.e. the instance IRIs and literal values appear in the constraints. This makes the constraints very dependent on the respective instances, hence the potential for reusing them is low. In our work, instead, we define constraints on the level of classes. Our constraints are highly reusable. In addition, in his work, no general feature tree ontology is specified. In our work, instead, we propose one.

5 Conclusion and Future Work

To tackle the problem of compliance management in regulated industries characterized by demanding variability management needs, we have proposed an ontology-based representation for representing feature trees as RDF graphs and reusable SHACL constraints to shape the RDF graphs. In future, we aim at conducting case study-based research considering the entire layered structure [4].

References

1. Allian, A.P., Oliveira Jr., E., Capilla, R., Nakagawa, E.Y.: Have variability tools fulfilled the needs of the software industry? J. Univ. Comput. Sci. **26**(10), 1282–1311 (2020)
2. Capilla, R., Gallina, B., Cetina, C., Favaro, J.: Opportunities for software reuse in an uncertain world: from past to emerging trends. J. Softw. Evol. Process **31**(8), 1–22 (2019)
3. Fleischhacker, P.: Validation of feature models using semantic web technologies. Master's thesis, Graz University of Technology, Graz, Austria (2021)
4. Gallina, B., Munk, P., Schweizer, M.: An extension of the rasmussen socio-technical system for continuous safety assurance. CARS 2024, Apr 2024, Leuven, Belgium. ⟨hal − 04558510⟩. https://hal.science/hal-04558510
5. Hofman, P., Stenzel, T., Pohley, T., Kircher, M., Bermann, A.: Domain specific feature modeling for software product lines. In: Proceedings of the 16th International Software Product Line Conference (SPLC)-Volume 1, pp. 229–238. Association for Computing Machinery, New York, NY, USA (2012)

6. Nešić, D., Krüger, J., Stănciulescu, U., Berger, T.: Principles of feature modeling. In: Proceedings of the 27th ACM Joint Meeting on ESEC/FSE, pp. 62–73, New York, NY, USA (2019)
7. UNECE: REGULATION 21 - Uniform Provisions Concerning the Approval of vehicles with regard to their interior fittings, April 2003
8. W3C: OWL 2 Web Ontology Language Manchester Syntax (2012)
9. W3C: SPARQL 1.1 Query Language (2013)
10. W3C: RDF 1.2 Concepts and Abstract Syntax (2014)
11. W3C: Shapes Constraint Language (SHACL), W3C Recommendation (2017)
12. Zaid, L.A., Kleinermann, F., De Troyer, O.: Applying semantic web technology to feature modeling. In: Proceedings of the ACM Symposium on Applied Computing (SAC), pp. 1252–1256. Association for Computing Machinery, New York, NY, USA (2009)

Adaptation and Reuse

Assessing Reflection Usage with Mutation Testing Augmented Analysis

Iona Thomas(✉), Stéphane Ducasse, Guillermo Polito, and Pablo Tesone

Univ Lille, Inria, CNRS, Centrale Lille, UMR 9189 - CRIStAL, Lille 59650, France
{Iona.Thomas,Stephane.Ducasse,Guillermo.Polito,Pablo.Tesone}@inria.fr

Abstract. Reflection is a powerful tool that allows a program to manipulate itself during its execution. However, developers may use it to circumvent data encapsulation and method visibility modifiers. Thus, it is important to assess how much an application relies on reflection.

Nonetheless, reflection is mostly incompatible with static analysis as it relies on runtime information (*e.g.,* to determine the attribute to be accessed or the method to evaluate). These problems worsen with dynamically-typed languages, where reflective operations are polymorphic with non-reflective operations, *e.g.,* in Pharo, array access is polymorphic with context variable modifications.

In this paper, we present RAPIM, an approach to study the uses of reflective APIs: it uses mutation analysis with a new mutation operator for dealing with core reflective methods. We analyze a serialization library from a developer perspective, showing the information it reveals. We evaluate our approach on a selection of five projects by comparing its performance against static analysis. We show that out of five projects, RAPIM disambiguates more potentially reflective call-sites than the static analysis. When the code coverage is good, the percentage of disambiguation is three times higher. Finally, we question the relevance of polymorphism between non-reflective and reflective APIs. Out of five projects, only one uses it, for only 1.4% of potentially reflective call-sites. We argue that reflective APIs could be renamed to avoid ambiguities.

1 Introduction

Reflective operations allow a program to manipulate itself (and its programming language) during its execution. This includes [17,22]:

- *Introspection*, the ability to examine its own structure and state,
- *Self-modification*, the ability to change itself,
- *Intercession*, the ability to alter the semantics of its programming language.

They are a powerful tool, allowing us to build debugging tools, refactorings, and new language features [10,24,28].

The Need for Reflection Usage Analysis. Although powerful, reflective features are usable as security exploits. For example, they allow malicious users to violate encapsulation and execute methods that were not intended to be

A. Achilleos et al. (Eds.): ICSR 2024, LNCS 14614, pp. 105–122, 2024.
https://doi.org/10.1007/978-3-031-66459-5_7

executed [11,19,23]. This means that *developers must assess how much they rely on reflection.* It is important to understand if a reflective functionality is central to an application or if it is only confined to peripheral parts (See Sect. 2.1).

Challenges of Reflection Analysis. These problems become exacerbated in dynamically-typed languages. On the one hand, reflective features defeat static analysis [5,15] because the actual attributes/methods that are being used are decided at runtime. On the other hand, reflective APIs are often designed as normal methods and are often polymorphic with non-reflective operations. For example, in Javascript reflectively accessing an attribute (`object['attribute']`) is syntactically equivalent to array/dictionary accesses (`dictionary[index]`).

> *Reflection has always been a thorn in the side of Java static analysis tools. Without a full treatment of reflection, static analysis tools are both incomplete because some parts of the program may not be included in the application call graph, and unsound because the static analysis does not take into account reflective features of Java that allow writes to object fields and method invocations. However, accurately analyzing reflection has always been difficult, leading to most static analysis tools treating reflection in an unsound manner or just ignoring it entirely. This is unsatisfactory as many modern Java applications make significant use of reflection* [15].

While the quote above is about Java, this tension is exacerbated in the case of deeply reflective languages such as Smalltalk descendants. Pharo, for example, as a descendant of Smalltalk is the essence of a reflective language with advanced reflective operations such as bulk pointer swapping [20], on-demand stack reification, and first-class resumable exceptions. In addition, in Smalltalk and many of its derivatives, reflective facilities are mixed with the non-reflective API of objects and classes [4,24,28]. They are a key part of the kernel of the language and libraries (See Sect. 2.2).

Mutation-Based Assessing Reflection Uses. In this paper, we propose to augment reflection security analysis with mutation analysis. We design reflection-specific mutations to obtain dynamic runtime information (See Sect. 3). We then use this information to assess the dependencies on reflective features and the degraded modes of an application (See Sect. 4).

Contributions

- First, an approach based on mutation testing to assess the dependencies of an application on certain reflective APIs.
- Second, RAPIM an implementation strategy to contextualize the analysis *i.e.,* handling the fact that reflective features are often a core part of a language (Python, Pharo...) and cannot be simply pulled off.
- A way to report results in a structured manner.
- An evaluation of the approach on a selection of projects. We show that in four out of five projects RAPIM disambiguates more *potentially reflective call-sites* than static analysis. When the coverage is good, the disambiguation rate was up to three times higher or more.

– An evaluation of the relevance of polymorphism between reflective and non-reflective APIs. We show that the polymorphism is very rarely used, with only one project leveraging it for 1.4% of its *potentially reflective call-sites*.

2 Background: Reflection Analysis

2.1 Why Assessing Reflection Dependencies

Developers need to monitor how much they rely on reflection without having to analyze the code manually. We want to provide an automated way to assess how much an application or library relies on reflective operations, and which ones. This is important because reflective operations

– allow one to bypass encapsulation and break invariants,
– can make the system unstable, and
– might introduce a performance cost.

When performing an analysis developers should not get drowned in a sea of information. Information should be presented with different levels of granularity and potentially convey more semantical information. Here are more precise questions to characterize a reflective feature usage.

– *General use.* How much does a project rely on reflection in general? The first general objective is to understand if an application uses or not reflective features.
– *Faceted analysis.* How much does it rely on specific reflective features? There are different families of reflective features: simple introspection, encapsulation violation, memory scanning, and more [28]. Each of such families has different consequences on security issues. This is why this is important to propose a faceted analysis.
– *Spatial distribution.* Which application parts are impacted by a given reflective use? During the assessment developers need to understand the spread of a reflective use.
– *Degraded modes.* How much of the application still works without a given reflective use?

Note this article is not about validating if our solution addresses the previous questions since it would involve a large user study. The present paper is about how can we perform the analysis on top of which such questions could be answered. However, Sect. 4 is an example providing a glimpse of what such an analysis could look like.

2.2 The Challenges of Reflection Analysis

Dynamic Decisions are a Blind Spot of Static Analysis. Reflective features defeat static analysis because the actual attributes/methods that are being used are decided and even crafted at runtime [12,13,15,23]. It is thus impossible to precisely know using static analyses what methods will be called by a reflective method invocation, or what fields will be read/written using a reflective field access.

Polymorphism Between Reflective and Non-reflective Methods. Moreover, such reflective dependency analysis is even more difficult in dynamically-typed languages where reflective APIs are often designed as normal methods that are polymorphic with non-reflective operations. As already mentioned, in Javascript and Python reflectively accessing an attribute (`object['attribute']`) is syntactically equivalent to array/dictionary accesses (`array[index]`/`dictionary[key]`). In Pharo, 44% of the reflective method's selectors have also non-reflective implementors.

This makes reflective operations look like any other message. Reflective features are handy but they should not be used by accident, and should not be mistaken for regular non-reflective messages. While solutions such as mirrors [6] offer a way to separate the base level from the meta-level, it requires the full redesign of some core functionalities of a language. This does not solve the problems of developers of existing languages.

3 RAPIM: Reflective API Mutation Analysis

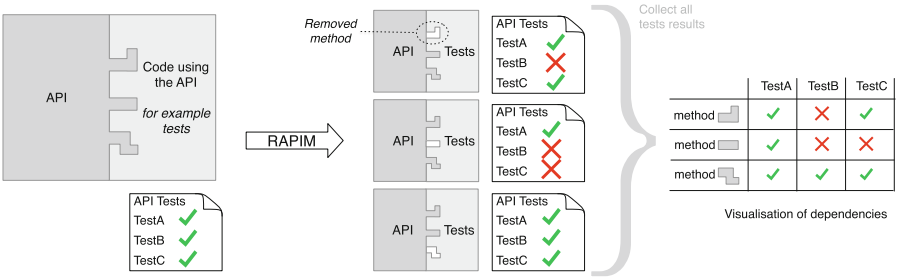

Fig. 1. RAPIM: Using mutation analysis to assess reflection usage. For each reflective method removed, all tests are run and results are collected.

Modern development methodologies such as Agile Programming and Test-Driven Design [1–3,18] advocate the systematic use of tests. Tests are an automated way to verify a sort of executable specification. While a suite of tests usually aims at testing various inputs, especially boundary values, and to cover as much code and paths as possible for the tested application, they are rarely randomly written but centered around the validation of application features. We propose to leverage these tests to assess dependencies to reflective operations.

Our approach, RAPIM, extends mutation analysis to assess reflective features use as illustrated by Fig. 1. Mutation testing is a technique that allows one to evaluate the coverage of a test suite by modifying the code and checking that at least one test break. Indeed if a test breaks, the test suite detects the modification. Here we remove reflective methods and we assess the impact by running the tests.

3.1 Terminology

This section introduces some terminology required to understand the rest of this article, supported by Fig. 2.

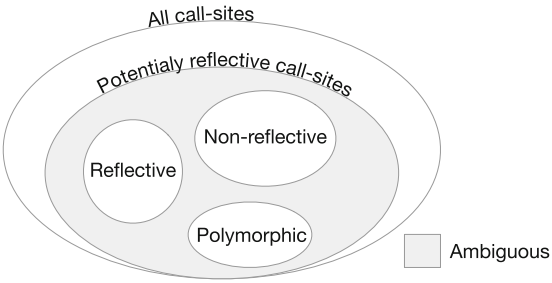

Fig. 2. Call-sites terminology

Potentially reflective call-site. A call-site is potentially reflective if it could directly call a reflective method (*i.e.,* not in a transitive manner). We consider that a call-site is potentially reflective if there is at least one reflective method implementing the call-site signature. For example, `at:put:` has at least one reflective implementor, therefore all `at:put:` call-sites are potentially reflective.

Reflective call-site. A *potentially reflective call-site* that only calls reflective methods.

Non-Reflective call-site. A *potentially reflective call-site* that only calls non-reflective methods.

Polymorphic call-site. A *potentially reflective call-site* that calls both reflective and non-reflective methods.

Ambiguous call-sites. A set of *potentially reflective call-sites* that we cannot identify as reflective, non-reflective or polymorphic.

3.2 Reflective Mutation Analysis Approach

Mutation testing works by applying a non-semantic-preserving mutation and checking if at least a test breaks after such a change. We extended Mutalk, a mutation testing framework for the Pharo programming language, and designed a reflection-specific mutation operator to assess the use of reflection, explained in Sect. 3.3. This operator works on pairs of call-site and reflective methods to identify which one is called, if any. We use the list of reflective methods defined by Thomas et al. [28]. We use this list to consider that a call site is a *potentially reflective call-site*.

Since we want to provide a full assessment of the reflective API use, we configured the mutation framework to run all the tests covering a mutation instead of stopping at the first one that failed. This allows us to report the percentage of working tests after a mutation. Our operator will then fail tests using reflective features and this allows us to identify per test:

– **Surviving Mutant.** A reflective feature that is not used by the test.
– **Killed Mutant.** A reflective feature is used by the failed test.

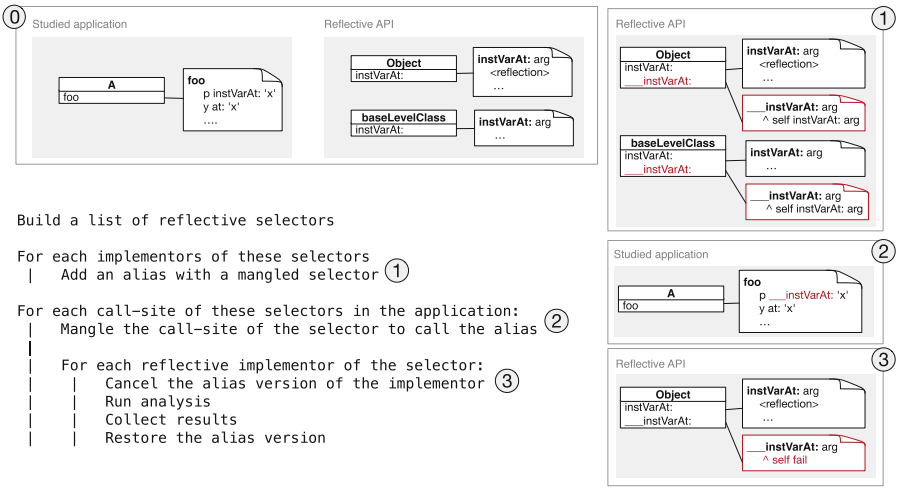

Fig. 3. Code transformation for *Reflective method canceling*.

3.3 Reflective Method Cancelling Operator

For the mutation analysis, we want tests to fail when a specific reflective method is called from a specific *potentially reflective call-site*. To cancel a reflective method (so that calling it makes the current test fail), it is not sufficient to rewrite its call-site to invoke a different (potentially failing) method. Canceling a call-site in such a way will indeed fail when calling reflective methods, but will also fail when calling non-reflective methods. Remember that in Pharo, for example, the message `at:put:` is both implemented by collections (non-reflective setter) and the execution stack (reflective). Call-sites calling `at:put:` on collection should not fail.

To analyze the usage of reflection, we designed a reflection-specific operator that cancels all pairs (reflective method, call-site), one by one, taking into account for each *potentially reflective call-site* all reflective methods that could be invoked by that call site. We call this the *reflective method canceling operator*, illustrated in Fig. 3. For each pair, the operator:

1. rewrites the *potentially reflective call-site* to an alias of the original method (see step 2, Fig. 3)
2. introduces a canceled alias for the method (see step 3, Fig. 3)

Then all tests covering this *potentially reflective call-site* are run. If the call-site calls the canceled reflective method the test fails, if it calls any other implementation, the code run normally, with only an additional layer of indirection.

Our implementation further optimizes this approach by pre-installing the aliases for all implementors of reflective selectors instead of doing it for each call-site (see step 1, Fig. 3).

4 RAPIM by Example: A Developer Perspective

In this section we illustrate RAPIM with a use case: Pharo's STON serialization framework. Through this use case, we show how it helps the developer to answer questions listed in 2.1. Table 1 shows a first glance at the distribution of *potentially reflective call-site* 13% of the call-sites are not covered by the tests. 26% of the call-sites are only calling reflective methods. 61% are only calling non-reflective methods.

Table 1. STON's *potentially reflective call-sites* classification with RAPIM

Percentage of:	Non covered	Refl.	Non-Refl.	Polymorphic
potentially reflective call-sites	13%	26%	61%	0

Figure 4 shows that out of the 13 selectors used by these call sites:

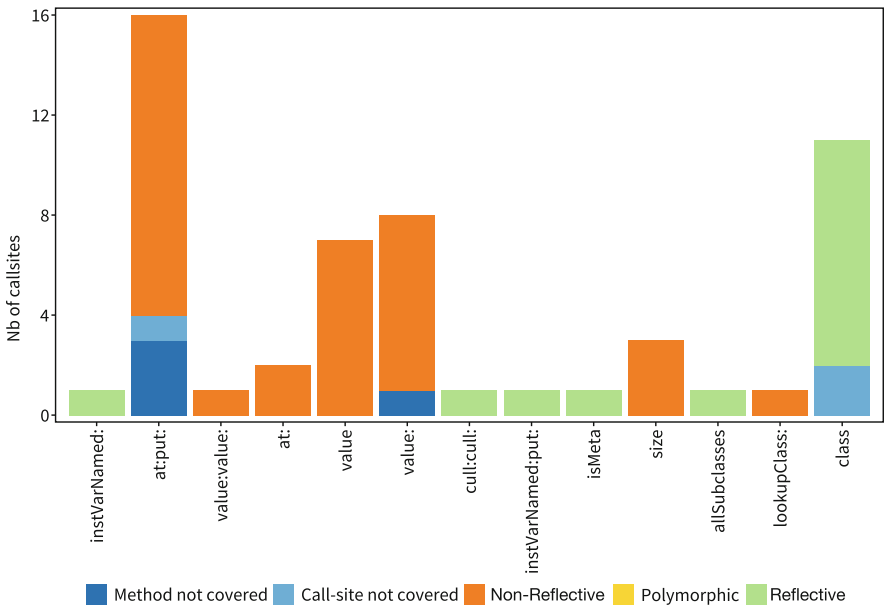

Fig. 4. Proportion of call sites types by selector for STON.

- 5 of them only have callsites calling non-reflective methods (#value:value: #at: #value #size #lookupClass)
- 5 of them only have callsites calling reflective methods (#instVarNamed: #cull:cull: #instVarNamed #isMeta #allSubclasses). Those have only one callsite each, which means that these uses of reflection are not too spread around in the application.
- #class has two out of eleven callsites that are not covered by tests. However, a static analysis reveals that #class has only one implementor. Therefore, all of those callsites are reflective too.
- All #at:put: and #value: covered call-sites are non-reflective, but some are not covered which means that we cannot conclude anything about them.

The matrix on Fig. 5 gives fine-grained information: the percentage of broken tests in test classes if we actually remove each reflective method. STON relies on seven reflective methods. MySTONCStyleCommentsSkipStreamTest is the only test class whose tests do not use reflection. Assuming serialization is tested by MySTONWriterTest, it only relies on #instVarNamed: and #class which was expected. For deserialization, MySTONReaderTest requires more reflective methods, as it relies on six methods (See Fig. 5 for the details).

Table 1, Figs. 4 and 5 highlight different levels of detail that can lead to a better understanding of the dependency of a project on reflection. The matrix visualization provides also other levels, without grouping the tests by class for

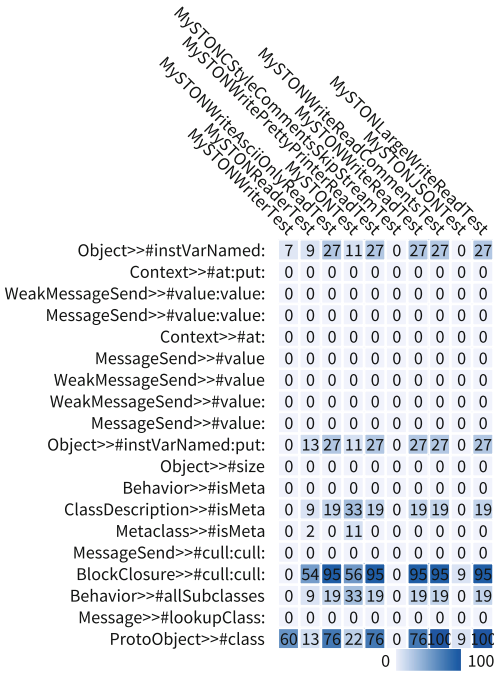

Fig. 5. Table of percentage of tests in a given class depending on a reflective method

finer information, or by grouping reflective methods by categories of reflective methods for coarser ones (See Appendix B).

5 Evaluation of Reflective API Identification

In the previous section, we reported how our analysis supports the developers in assessing the dependencies of the system to reflective features. In this section, we answer the research questions that drove our experiments:

- RQ1. Mutation Comparison Does mutation analysis improve the detection of reflective API usages in comparison to static analysis?
- RQ2. Effective Polymorphism Is the polymorphism between non-reflective and reflective methods actually used by applications? Could we safely rename reflective methods to disambiguate *potentially reflective call-sites*?

5.1 Chosen Projects

Following is the list of the projects we studied, and the criteria for selection of each of them. Table 2 summarizes the results of applying RAPIM on these projects. (See Appendix A for links to repositories)

- *STON.* SmallTalk Object Notation. A textual object serializer/deserializer inspired by JSON. We expect reflection to be used for reading class information and instance variables for serialization and setting instance variables for deserialization.
- *Microdown.* A parser for a markup language derived from Markdown [8]. We do not expect many reflection uses in this application.
- *Refactoring.* The refactoring framework used in production by the Pharo development environment [25,26]. We studied both the *Refactoring-Core* and the *Refactoring-Transformations* packages. We expect a heavy use of reflection as it is manipulating methods and classes to perform code rewritings.
- *MuTalk.* A mutation testing framework, the same we use for this analysis. We worked on a copy of these packages to be able to execute our approach on its framework. We expect the use of reflection to edit methods to install mutations.
- *Seaside.* A web application framework maintaining sessions using continuations [9]. We expect the use of reflection in the continuation management.

Table 2. Projects statistics and their *potentially reflective call-site* classification with RAPIM

Project	version	#classes	#tests	#call-sites	Non-cov.	Refl.	Non-Refl.	Poly.
STON	bbe8f5f	14	310	54	13%	26%	61%	0
Microdown	72f4ac7	168	543	576	77%	3%	20%	0
Refactoring	Pharo12 build.1386	211	807	1022	39%	7,4%	52,2%	1,4%
MuTalk	e712ac5	150	303	238	48%	11%	41%	0
Seaside	56286ac	547	876	1314	74%	5%	21%	0

5.2 Answering RQ1. Mutation Comparison

Does Mutation Analysis improve the detection of reflective API usages in comparison to static analysis?

Static analysis classifies *potentially reflective call-sites* according to the implementors of the method called. If all implementors are reflective, the call-site is identified as a reflective call-site. Otherwise, if there is at least one non-reflective implementor, the call-site is ambiguous.

Table 3 shows the number of *potentially reflective call-sites* that static analysis identifies as reflective or ambiguous, and whether they are covered by tests or not. As explained in Sect. 3.1, reflective call-sites have only reflective implementations, and ambiguous call-sites' implementors have a subset that is reflective and a subset that is non-reflective.

Table 3. Number of *potentially reflective call-sites* by projects, split by coverage and reflective ambiguity according to static analysis.

Project	Static analysis	Not Covered	Covered
STON	Statically Reflective	2 *(4%)*	11 *(20%)*
	Statically Ambiguous	5 *(9%)*	36 *(67%)*
Microdown	Statically Reflective	174 *(30%)*	16 *(3%)*
	Statically Ambiguous	272 *(47%)*	114 *(20%)*
Refactoring	Statically Reflective	108 *(10.5%)*	32 *(3%)*
	Statically Ambiguous	292 *(28.5%)*	590 *(58%)*
MuTalk	Statically Reflective	20 *(8%)*	15 *(6%)*
	Statically Ambiguous	95 *(40%)*	110 *(46%)*
Seaside	Statically Reflective	197 *(15%)*	52 *(4%)*
	Statically Ambiguous	779 *(59%)*	286 *(22%)*

24% disambiguated by static analysis

87% disambiguated by RAPIM

Static analysis can only identify reflective call-sites for which all implementors are reflective. On the other hand, RAPIM analysis scope only includes covered call-sites. Ambiguous and covered *potentially reflective call-sites* are analyzed by RAPIM to identify the reflective ones. Non-covered reflective call-sites are not detected by the dynamic analysis of RAPIM because this approach is based on code coverage. The ambiguous and not covered call-sites are the ones neither approach can disambiguate.

As shown in Table 3, all projects have ambiguous *potentially reflective call-site* that are covered by tests. Those call sites cannot be disambiguated by static analysis, but RAPIM allows one to classify them. For example in the STON project, 87% of the *potentially reflective call-sites* are disambiguated by RAPIM, and 24% of the *potentially reflective call-sites* are identified as reflective by the static analysis (See annotation on Table 3). Our analysis shows that 67% of the *potentially reflective call-sites* are ambiguous and covered. The 20% of statically reflective and covered call-sites are identified by both the static analysis and RAPIM. The 4% of not covered and statically reflective call sites are not taken into account by the dynamic analysis. Conversely, the 67% of ambiguous and

covered call-sites are not disambiguated by the static analysis. The 9% remaining not covered and ambiguous call-sites are the ones neither approaches can disambiguate. The projects Refactoring and MuTalk have similar profiles.

The situation is slightly different in Microdown where the coverage of *potentially reflective call-sites* is lower (23%). There is only 3% of covered and reflective call-sites. This means that the overlap between static analysis and RAPIM is small. RAPIM disambiguates 20% of the *potentially reflective call-sites*. This is the lowest among the five studied projects. 47% of the call sites are never disambiguated. In addition, the static analysis disambiguates 33% of *potentially reflective call-sites* (which is the highest among our 5 projects) while RAPIM only 23%. This tilts the balance in favor of static analysis for this projects. The Seaside project exhibits the same profile, but the lower amount of statically reflective call-sites gives RAPIM the advantage.

Conclusion. On four out of five projects, RAPIM disambiguates more *potentially reflective call-sites* than the static analysis. When projects have good coverage (STON, MuTalk, Refactoring), our approach disambiguates **three times more** *potentially reflective call-sites* than the static analysis. For projects with low coverage, the disambiguation is on par with static analysis, but they present a high percentage of call sites that are still ambiguous.

5.3 Answering RQ2. Effective Polymorphism

Is the polymorphism between non-reflective and reflective methods used by applications? Could we safely rename reflective methods to disambiguate potentially reflective call-sites*?*

Table 2 shows that only one project uses polymorphism between reflective and non-reflective methods. This is expected because the refactoring engine has its meta-model of the code that mimics part of Pharo's reflective API [28]. Moreover, it mixes Pharo meta-objects with its own code model. Even in this case, Table 2 shows that polymorphic usage concerns only 1.4% of the *potentially reflective call-sites*.

Conclusion. Out of the five projects, only the refactoring engine leverages the polymorphism between reflective and non-reflective APIs. Even in this case, this is a very rare usage (1.4% of *potentially reflective call-sites*). This is **strong evidence** showing that reflective APIs could be re-designed to be mostly non-ambiguous in dynamically-typed languages.

6 Discussion

This section discusses the limitations of our approach and the threats to validity of our evaluation.

6.1 Limitations

Code Coverage. RAPIM cannot disambiguate *potentially reflective call-sites* that are not covered. We saw in Sect. 5 that we get better results with a higher coverage. This is a limitation inherited from mutation analysis in general. However, our evaluation shows that when tests are available, mutation analysis complements static analyses.

Other Meta-object. RAPIM only identifies dependencies to reflective methods. It does not cover access to global variables or stack reifications with the pseudo-variable thisContext. However, many methods that could be called on thisContext are identified as reflective methods (*i.e.,* while accessing to the reified stack is not detected by RAPIM, the use of many methods on the stack reification is detected).

Indirect Dependencies. If the studied application relies indirectly on reflection (*i.e.,* it uses a library that uses reflection), these dependencies will not be identified. RAPIM is designed to detect direct calls from the studied application to the reflective API scoped to a package.

Studying Core Packages. RAPIM cannot be used directly on the standard libraries of the Pharo programming language, as it could break the system. As in the case of MUTALK, this can be solved by creating copies of the studied packages and running RAPIM on this copy.

6.2 Threats to Validity

Internal Validity.

Code Coverage. Studying real-life application is necessary to evaluate effective polymorphism, but our selection comes with a wide range of code coverage. Our analysis relies on the fact that tests cover both reflective and non-reflective cases. Having some projects with higher coverage mitigates the risk that reflective cases are overlooked in tests. The fact that we do not have a higher polymorphism percentage in those projects supports our conclusion.

Pharo Exception Handling. We left out of the analysis several tests (13) related to exception handling in the Seaside project. Running RAPIM on Seaside introduced a bug leading to infinite loops in these tests. The impact of removing thos is mitigated by the amount of other tests as removed tests only represent 1,5% of the total tests one this project.

Construct Validity.

Studying Core Packages. RAPIM cannot be used directly on the core libraries of the Pharo programming language, as it could break the system. To study such packages (*e.g.,* STON), we duplicate the package and run RAPIM on the copy. To ensure that results on the copy would be informative about the original package, we made sure that the duplicated version still runs well, and that all its tests are green.

External Validity.

Project Selection for the Evaluation. The evaluation of our approach relies on the results of RAPIM on five applications. To run this evaluation, we chose a set of various projects that we were expecting to use different amount and features of reflection. We specifically aimed for variety to mitigate the bias introduced by selecting only a few projects.

7 Related Work

This section presents related work in two different axes. First, the existing approaches to overcome the limitations of using only static analysis in dynamic languages. Then, the usage of mutation testing for program analysis and validation.

7.1 Overcoming the Limits of Static Analyses

Ruf [27] proposes the use of partial evaluation to detect the meta-level operations, once these meta-level operations are detected they are replaced by equivalent code as they are not present at run-time. Braux and Noyé [7] improve the results of static analysis for Java programs by applying partial evaluation to reflection resolution to apply optimizations. Their paper describes extensions to a standard partial evaluator to offer reflection support. The idea is to *compile away* reflective calls in Java programs, turning them into regular operations on objects and methods, given constraints on the concrete types of the object involved. The type constraints for performing specialization are manually provided.

Tip *et al.,* propose Jax [29]: an application extractor and compactor for Java. In their solution, Jax performs a static analysis of the program and builds a call graph of the application. It then removes unused methods and compacts the application. It is also affected by the limitations of static analysis. To handle the missing information the authors propose three alternatives: (1) it requires user intervention to handle the dynamic loading and execution of code, (2) it performs a conservative selection of possible methods for a given call site, and (3) assumes that all methods in external library interfaces are called.

Bodden *et al.,* [5] approaches the limitations of static analysis by integrating static analysis tools with runtime information. Their tool inserts runtime checks

into the code. These checks warn the user in case the program is performing reflective calls that were not identified by the static analysis. More recently, Liu *et al.,* [14] improved the approach of Bodden *et al.,* by taking benefit from runtime information obtained from code coverage. They automatically generate test cases to improve the code coverage and the detection of reflective call targets. Similar to us, this work uses runtime information, and more specifically tests, to augment static analysis. However, their objective differs: while they intend to obtain information on reflective call targets, our main goal is to understand the usage of reflective operations and de-ambiguate reflective from non-reflective operations in dynamic languages.

7.2 Program Analysis and Validation via Mutation Testing

Mouelhi *et al.,* [21] propose using mutation testing to detect security issues. Loise *et al.,* [16] based on this idea propose a series of mutation operators to detect specific security issues. They propose 15 mutation operators that are applied to Java programs. Using this technique they detect security issues that are ignored by static analysis. Our solution is not designed specifically for detecting security issues, but it is possible to identify misuses of reflective calls that introduce them.

Wen *et al.,* [30] propose using mutation testing to detect misuses of library APIs. They represent API misuses as mutation operators applied in the code base. Then the mutant killing tests and their associated stack traces are collected to detect API misuses. We approach the identification of reflective usage in a similar way showing that a single mutation operator is enough for reflective usage analysis .

8 Conclusion

In this article we presented RAPIM, a mutation analysis approach based on a new mutation operator to understand dependencies to reflective APIs. This approach handles the fact that reflective features are often a core part of a language (Python, Pharo...) and cannot be simply pulled off. We then present with several levels of detail the identified dependencies for STON, the object serialization library used in Pharo.

To evaluate our approach, we used RAPIM on five different projects relying on reflection. On four out of five projects, RAPIM disambiguates more *potentially reflective call-sites* than the static analysis. When projects have good coverage (STON, MuTalk, Refactoring), our approach disambiguates three times more *potentially reflective call-sites* than the static analysis. For projects with low coverage, the disambiguation is on par with static analysis, but they present a high percentage of call sites that are still ambiguous.

Out of the five projects, only the refactoring engine leverages the polymorphism between reflective and non-reflective APIs. Even in this case, this is a very rare usage (1.4% of *potentially reflective call-sites*). This supports the idea

that reflective API could be renamed to avoid accidental polymorphism with non-reflective methods (Figs. 6, 7)

We believe this approach applies to domains other than reflection and could help to sort out critical dependencies and further security analysis, such as dependencies to file access APIs or user inputs (Table 4).

Acknowledgement. We would like to thank the anonymous reviewers for their useful feedback. We are also grateful to the European Smalltalk User Group (http://www.esug.org) for their financial support.

A URLs to Studied Projects

Table 4. Links to project repositories used for this analysis

Project	Url Link
STON	https://github.com/svenvc/ston
Microdown	https://github.com/pillar-markup/Microdown
Refactoring (a part of Pharo)	https://github.com/pharo-project/pharo
MuTalk	https://github.com/pharo-contributions/mutalk
Seaside	https://github.com/SeasideSt/Seaside

B Additional STON Matrices

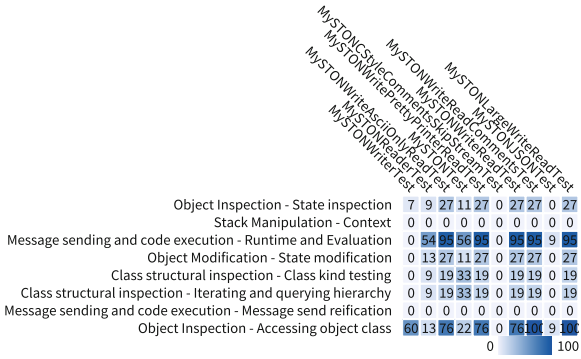

Fig. 6. Table of percentage of tests in a given class depending on a reflective category

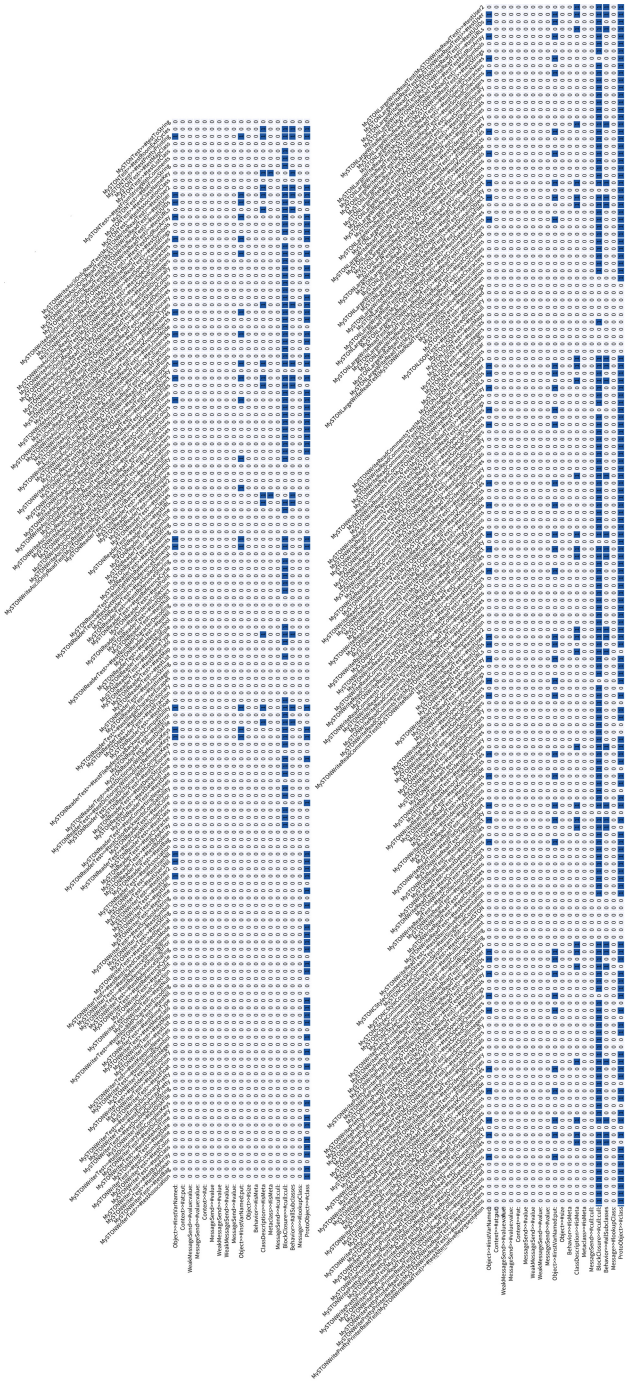

Fig. 7. Table of Tests depending on a reflective method. A 1 (blue cell) means that the test has failed when this reflective method is removed. (Color figure online)

References

1. Astels, D.: Test-Driven Development — A Practical Guide. Prentice Hall (2003)
2. Beck, K.: Manifesto for agile software development. http://agilemanifesto.org
3. Beck, K.: Extreme Programming Explained: Embrace Change. Addison Wesley (2000)
4. Black, A.P., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., Denker, M.: Pharo by Example. Square Bracket Associates, Kehrsatz, Switzerland (2009), http://books.pharo.org
5. Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., Mezini, M.: Taming reflection: aiding static analysis in the presence of reflection and custom class loaders. In: Proceedings of the 33rd International Conference on Software Engineering,ICSE 2011, pp. 241–250. ACM, New York (2011). https://doi.org/10.1145/1985793.1985827
6. Bracha, G., Ungar, D.: Mirrors: design principles for meta-level facilities of object-oriented programming languages. In: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004), ACM SIGPLAN Notices, pp. 331–344. ACM Press, New York, (2004). http://bracha.org/mirrors.pdf
7. Braux, M., Noyé, J.: Towards partially evaluating reflection in java. ACM SIGPLAN Not. **34**(11), 2–11 (1999)
8. Ducasse, S., Dargaud, L., Polito, G.: Microdown: a clean and extensible markup language to support pharo documentation. In: Proceedings of the 2020 International Workshop on Smalltalk Technologies (2020)
9. Ducasse, S., Renggli, L., Shaffer, C.D., Zaccone, R., Davies, M.: Dynamic Web Development with Seaside. Square Bracket Associates (2010). http://book.seaside.st/book
10. Forman, I.R., Forman, N.: Java Reflection in Action (In Action series). Manning Publications Co., USA (2004)
11. Hunt, G.C., Larus, J.R.: Singularity: rethinking the software stack. SIGOPS Oper. Syst. Rev. **41**(2), 37–49 (2007). https://doi.org/10.1145/1243418.1243424
12. Landman, D., Serebrenik, A., Vinju, J.J.: Challenges for static analysis of java reflection - literature review and empirical study. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 507–518 (2017https://doi.org/10.1109/ICSE.2017.53
13. Li, S., Dietrich, J., Tahir, A., Fourtonis, G.: On the recall of static call graph construction in practice. In: ICSE (2020)
14. Liu, J., Li, Y., Tan, T., Xue, J.: Reflection analysis for java: uncovering more reflective targets precisely. In: 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), pp. 12–23 (2017). https://doi.org/10.1109/ISSRE.2017.36
15. Livshits, B., Whaley, J., Lam, M.S.: Reflection analysis for java. In: Proceedings of Asian Symposium on Programming Languages and Systems (2005)
16. Loise, T., Devroey, X., Perrouin, G., Papadakis, M., Heymans, P.: Towards security-aware mutation testing. In: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 97–102 (2017). https://doi.org/10.1109/ICSTW.2017.24
17. Maes, P.: Concepts and experiments in computational reflection. In: Proceedings OOPSLA 1987, ACM SIGPLAN Notices. vol. 22, pp. 147–155 (Dec 1987). https://doi.org/10.1145/38807.38821

18. Martin, R.C.: Clean code: a handbook of agile software craftsmanship. Pearson Education (2009)
19. Miller, M.S., Samuel, M., Laurie, B., Awad, I., Stay, M.: Caja safe active content in sanitized javascript. Tech. rep., Google Inc. (2008). http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf
20. Miranda, E., Béra, C.: A partial read barrier for efficient support of live object-oriented programming. In: International Symposium on Memory Management (ISMM 2015), Portland, United States, pp. 93–104 (Jun 2015). https://doi.org/10.1145/2754169.2754186
21. Mouelhi, T., Le Traon, Y., Baudry, B.: Mutation analysis for security tests qualification. In: Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007), pp. 233–242. IEEE (2007)
22. Paepcke, A.: User-level language crafting. In: Object-Oriented Programming: the CLOS perspective, pp. 66–99. MIT Press (1993)
23. Richards, G., Hammer, C., Burg, B., Vitek, J.: The eval that men do: a large-scale study of the use of eval in javascript applications. In: Proceedings of Ecoop 2011 (2011)
24. Rivard, F.: Smalltalk: a reflective language. In: Proceedings of REFLECTION 1996, pp. 21–38 (Apr 1996)
25. Roberts, D., Brant, J., Johnson, R.E.: A refactoring tool for Smalltalk. Theory Pract. Object Syst. (TAPOS) **3**(4), 253–263 (1997)
26. Roberts, D., Brant, J., Johnson, R.E., Opdyke, B.: An automated refactoring tool. In: Proceedings of ICAST 1996 (Apr 1996)
27. Ruf, E.: Partial evaluation in reflective system implementations. In: Workshop on Reflection and Metalevel Architecture (1993)
28. Thomas, I., Ducasse, S., Tesone, P., Polito, G.: Pharo: a reflective language - A first systematic analysis of reflective APIs. In: IWST 23 - International Workshop on Smalltalk Technologies, Lyon, France (Aug 2023).https://inria.hal.science/hal-04217271
29. Tip, F., Laffra, C., Sweeney, P.F., Streeter, D.: Practical experience with an application extractor for java. In: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 1999, pp. 292-305. Association for Computing Machinery, New York (1999https://doi.org/10.1145/320384.320414
30. Wen, M., Liu, Y., Wu, R., Xie, X., Cheung, S.C., Su, Z.: Exposing library api misuses via mutation analysis. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 866–877 (2019).https://doi.org/10.1109/ICSE.2019.00093

Using Energy Consumption for Self-adaptation in FaaS

Pablo Serrano-Gutierrez[1,2(✉)] and Inmaculada Ayala[1,2]

[1] Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Malaga, Spain
{pserrano,ayala}@lcc.uma.es
[2] ITIS Software, Universidad de Málaga, Malaga, Spain

Abstract. One of the programming models that has been developing the most in recent years is Function as a Service (FaaS). The growing concern over data centre energy footprints has driven sustainable software development. In serverless applications, energy consumption depends on the energy consumption of the application's functions. However, measuring energy proves challenging, and the results' variability complicates optimisation efforts at runtime. This article addresses this issue by measuring serverless function energy consumption and exploring integration into an optimisation system that selects implementations based on their current energy footprint. For this, we have integrated an energy measurement software into a FaaS system. We have analysed how to properly process the data and how to use them to perform self-adaptation. We present a series of methods and policies that make our system not only capable of detecting variations in the energy consumption of the functions, but it does so taking into account the variability in the measurements that each function may present. Our experiments showcase proper integration in a self-adaptive system, showing a reduction up to 5% in energy consumption due to functions in a test application.

Keywords: sustainability · serverless · self-adaptive

1 Introduction

The growing concern about the energy footprint of datacentres [3,11] has encouraged the development of more sustainable software. Green Software engineering is a field that pursues software development and deployment more sustainably [8]. One of the most significant challenges in the field is measuring the energy consumption of the software. It can be measured with both hardware and software tools. Hardware tools provide greater accuracy but, on the other hand, are more expensive and challenging to implement. Furthermore, it is tough to isolate the energy consumption of a particular piece of software or process using this kind of solution because it measures the energy consumption of the whole system. Regarding software tools, we can find two types, some based on estimates

A. Achilleos et al. (Eds.): ICSR 2024, LNCS 14614, pp. 123–134, 2024.
https://doi.org/10.1007/978-3-031-66459-5_8

and others using hardware sensors. The latter has been developed recently since these hardware elements began to be incorporated into microprocessors just a decade ago.

Usually, to try to reduce the consumption of an application, optimisations are carried out or different configurations are tested, and for each case energy measurements are made at a global level to determine if consumption is reduced. The use of software tools allows us to not only measure overall consumption but also portions of the application, which can be used to optimise the different parts of it separately. Nevertheless, this is difficult to do in a traditional application, especially at runtime. However, due to their nature, serverless applications or those based on microservices are especially suitable for being optimised in parts.

Function as a Service (FaaS) environments are increasingly used since they save infrastructure maintenance and management costs. Serverless frameworks use virtual machines or containers to host serverless functions, which allows functions programmed in different programming languages to be used in the same application. This feature facilitates the integration of different work teams and software reuse. In addition, these frameworks support the automatic horizontal and vertical scaling of serverless functions based on demand.

Often, it is possible to implement a function in different ways. Also, when software is reused, we can have several implementations that suit the needs of the application. Each of these may be better suited to a given infrastructure or may have different performance from the point of view of user experience. For example, it is possible that an implementation is very energy efficient but needs a lot of available memory to work properly, so, in situations where memory is limited, it is possible that another implementation that initially was less energy efficient but does not need to use as much memory, behave better. So, an interesting approach to try to reduce consumption is to choose which of these functions are most appropriate to run in our infrastructure in terms of energy consumption.

Energy consumption of serverless applications is dynamic and challenging to predict at design time [5]. It depends on the infrastructure where the container is deployed and the execution conditions. In this work, we present the results of our experiments on energy saving for serverless applications. We have integrated an energy measurement software tool, Scaphandre[1], in a framework for the self-adaptation of serverless applications, and show how to use energy measurements to save energy of the serverless application at runtime. For this, we have defined procedures and rules that can be adapted to the necessities of an application. By executing the test cases, we verify that the system has self-adaptive behaviour, reducing the consumption of the application due to functions up to 5%.

This work is structured as follows: Sect. 2 presents some related work; Sect. 3 explains how we have integrated Scaphandre in the framework; Sect. 4 illustrates our approach for the analysis of energy measurements; Sect. 5 shows how to use it at runtime for energy saving; Sect. 6 presents the case study; and the paper finishes with some conclusions and ideas of future work.

[1] https://github.com/hubblo-org/scaphandre.

2 Related Work

Several studies analyse energy consumption in serverless systems. Moreno et al. [9] investigate the impact of cold-start techniques on response time and energy consumption, suggesting extending serverless platforms to the edge for latency reduction. Alhindi et al. [1] evaluate power efficiency of OpenFaaS [6] compared to Docker containers, highlighting energy efficiency of OpenFaaS in certain scenarios.

Other research focuses on energy-aware placement and scaling of serverless functions. Tsenos et al. [12] propose an Energy Efficient Scheduler to minimise energy consumption while meeting performance demands. FADE [13] is a tool proposed by Tzenetopoulos et al. that decomposes applications into serverless functions, considering energy consumption prediction for efficient node selection.

Other studies consider the architecture of the system. MicroFaaS [2] suggests energy-efficient architectures, utilising single-board computers to increase FaaS function energy efficiency by a factor of 5.6. Jiaxuechao et al. [5] analyse energy consumption breakdown in serverless computing, introducing energy fungibility and demonstrating its feasibility with a framework that can reduce function energy consumption up to 21.2%.

In conclusion, current work does not analyse the power consumption of functions individually and does not address consumption reduction based on the use of different implementations, which can be important in software reuse, as functions that have not been specifically designed to work on a specific system are often used.

3 Integrating Energy Measurement into FaaS

The solution presented in this section uses OpenFaaS [6], a framework for serverless functions, and Scaphandre, a tool to measure energy in containerised applications. We have selected OpenFaaS because it is a free-to-use serverless framework and energy efficient [1], but the approach can be easily adapted to other serverless frameworks.

As we have stated, Scaphandre is an energy measurement software tool which is currently under development. This tool bases its operation on RAPL (Running Average Power Limit) [4], a hardware element whose function is to limit consumption and which can be used as a source for these measurements. RAPL generates information about the system's energy consumption on a cumulative basis, and Scaphandre samples it and extracts information. As we know, $Energy{=}Power{\cdot}time$, so it is easy for Scaphandre to calculate the power consumed by the system from the energy samples. However, to obtain the energy consumed by applications or a process, the procedure becomes complicated since there is no way for RAPL to know which process each instruction being executed on the microprocessor belongs to. Scaphandre performs this task using jiffies, which are the CPU ticks used by the application. However, the calculation is more complex since it must consider other parameters, such as the core on which it is executed.

OpenFaaS deploys Docker[2] containers for each function and orchestrates them using Kubernetes[3]. Therefore, to determine the consumption of the serverless functions, we must know the consumption of the associated containers. Scaphandre can obtain measurements per process, allowing us to get the consumption of serverless functions. Each replica has an associated container corresponding to a running process. The information that Scaphandre returns about the processes is the power in microwatts (*scaph_-process_power_-consumption_microwatts*). However, to compare the energy consumption of the functions, the metric of interest is energy. Two functions can consume the same power, but it consumes more energy if one takes longer to execute. Therefore, our solution also measures the execution times of the functions to estimate their energy.

Scaphandre has numerous exporters through which we can read the data it provides. In our approach, we use Prometheus[4], a powerful software used extensively for containerised applications and also used by OpenFaaS. We configure Scaphandre to serve the data to Prometheus. To do this, we made a custom installation of OpenFaaS, setting Scaphandre as an additional data source.

Scaphandre can be installed on Kubernetes and has an option called *containers* that links container names with the energy measurements of the processes facilitating their identification. This information is obtained from the data contained in *proc/PID/cgroup* directory of the container structure. However, this is not possible for containers created by OpenFaaS. There, we had two options to get energy measurements: we could modify these containers to contain the appropriate information or search for the container manually. We opted for the second option because it facilitates this solution's future integration with other FaaS frameworks. The search procedure is as follows. Firstly, we obtain the identifier of a container associated with the function of interest to us by querying Kubernetes. Subsequently, we obtain the identification of the process associated with the said container, i.e., the *PID*, and, finally, we make a query to Prometheus with said *PID*. This procedure must be repeated for each container associated with each function replica. Adding the energy consumption of all the replicas, we will obtain the consumption of the function.

4 Analysing Energy Measurements

Before using the Scaphandre measurements to save energy, it is necessary to analyse the data obtained. After measuring the energy consumed by serverless functions, we observed significant variability. Suppose we want to compare the energy consumed by two implementations statically. In that case, it is not a problem since we can perform measurements over long periods and assign average values to each. However, if we want the system to react to changes that may

[2] https://www.docker.com/.
[3] https://kubernetes.io/.
[4] https://prometheus.io/.

Table 1. Serverless functions analysed

Function	Description	CPU usage	Memory usage	Ext. requests
f1	Get node info	Low	Low	No
f2	Text printing	Low	Medium	No
f3	Mathematical calculations	High	Low	No
f4	Search and manipulation of data	Medium	Medium	Yes
f5	Data compression	High	Medium	Yes
f6	OpenCV image analysis	High	High	Yes

occur in consumption over time, what interests us is that it can do so in a short period of time.

The fluctuations in serverless functions' energy consumption behaviour can cause unnecessary application adaptations. In this context, comparisons with alternate implementation become even more challenging. To avoid this, we can use several methods. One consists of waiting for the energy values to remain at a certain level for a time. That is, if the measurement presents a peak, it would not be taken as the new energy level, but rather, it should remain stable at that new value to be considered the new level of energy consumed. This presents the problem that peaks in the opposite direction can prolong the waiting time. Another method is calculating the average value of all the energy samples from the beginning. This has the problem that the longer the system has been active, the longer it will take for a change in the measured levels to be reflected. That is why moving averages are more useful. In this way, the n last samples obtained during a certain period of time are considered. Depending on the chosen period, we can smooth the energy curve more or less. However, we will also increase the time necessary to calculate, limiting our system's reaction time.

We have conducted tests with different serverless functions to analyse the functions' energy behaviour. The objective is to be able to check if the range of variation of the measurements is different depending on what the function performs. We have used six functions that perform calculations of different complexity, use different amounts of memory, and work locally or make external requests. Disk or database access should not be considered among these parameters since serverless functions do so through external requests. These functions are presented in Table 1. We have chosen them so that they are representative of the different types of functions considering the different combinations of values for these parameters, ranging from the simplest (*f1*) to the most complex (*f6*). Logically, other functions would have a different energy consumption, but what we are interested in is studying if there are differences in the variability in energy consumption depending on the type of processing that the function performs. We must keep in mind that our objective is to compare behaviours; we are not interested in specific energy measurements since these may vary depending on the infrastructure in which we deploy the application, the system load, etc.

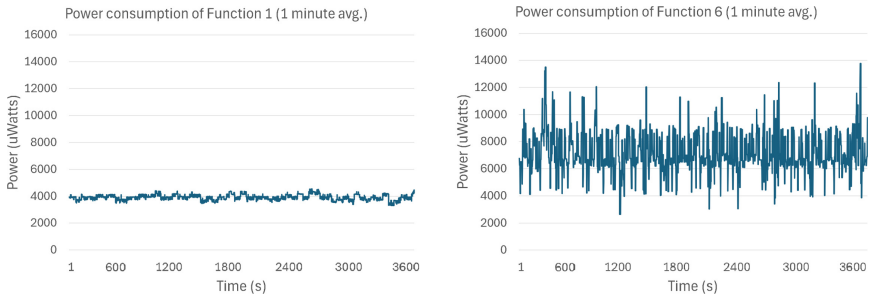

Fig. 1. Power consumption measurements obtained for *f1* and *f6*

The results obtained show that there are differences in the variability of power consumption measurements for each function. The simplest ones present less variability, while the more complex ones that include external requests are more variable. To analyse the differences, we will focus on those that present the least and greatest variability (see Fig. 1). We can decrease the variability by calculating the moving averages for more extended periods, as shown in Fig. 2.

We could evaluate an energy objective function for the application to trigger the system's self-adaptation process. However, to be able to react to changes in functions individually, it is more convenient to introduce action policies based on the energy consumed by each function. Once the policy is activated, the optimisation process determines the optimal configuration of the system at that moment. Thus, we can consider a threshold to trigger the self-adaptive process. Since the energy consumed by the functions is variable, it is most appropriate to define them in percentage. The chosen value must be greater than the maximum level of variability of the energy measurement we use since, otherwise, the normal variation of the measurements would unnecessarily launch a self-adaptation process. The maximum variation percentages of the mentioned functions are shown in blue in Fig. 3.

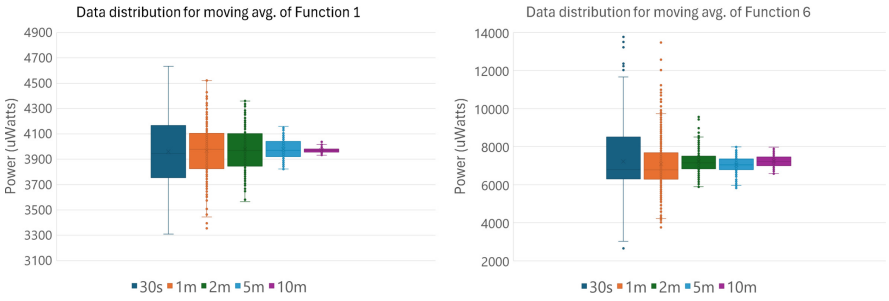

Fig. 2. Moving averages of *f1* and *f6* considering different periods

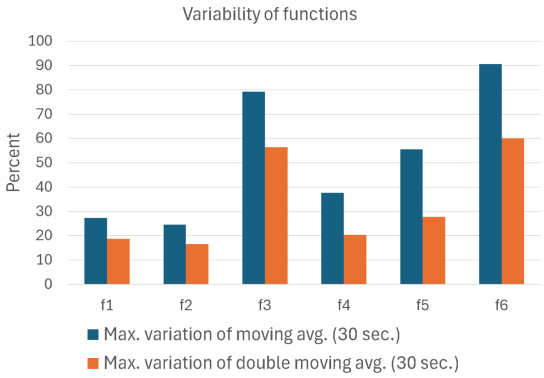

Fig. 3. Variation from average values of the test functions

In the case of the most variable function, the percentages are pretty high, so it would be interesting to improve the measurement method. To do this, we propose using the double moving average, a technique used in other fields of statistical analysis, such as financial analysis, in which it is used to study trends. The results are interesting since they notably smooth the energy curve (see Fig. 4). We can appreciate the reduction in variability by comparing them with the results for the moving average (Fig. 2) of the same functions. Besides, the results are improved compared to the moving average applied for the same total period of time. That is, the percentage variation of the double moving average for t is better than the moving average for $2t$. For example, in our tests, the maximum variation of the moving average of *f6* for 10 min was 9.6%, and the maximum variation of the double moving average for 5 min was 7.7%. The new maximum variation percentages of the tested functions are shown in orange in Fig. 3.

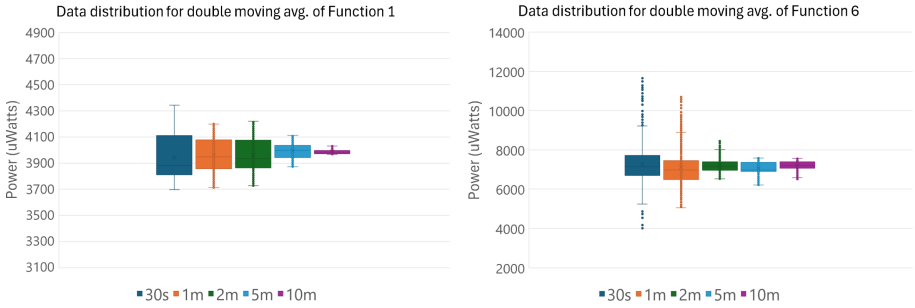

Fig. 4. Double moving averages of *f1* and *f6* considering different periods

5 Building the Self-adaptive System

To build our self-adaptive system, we use a framework we previously developed [10]. It is designed to find optimal configurations by considering quality of service requirements. This framework is integrated with OpenFaaS and Scaphandre to constitute our complete system, whose architecture is detailed in Fig. 5.

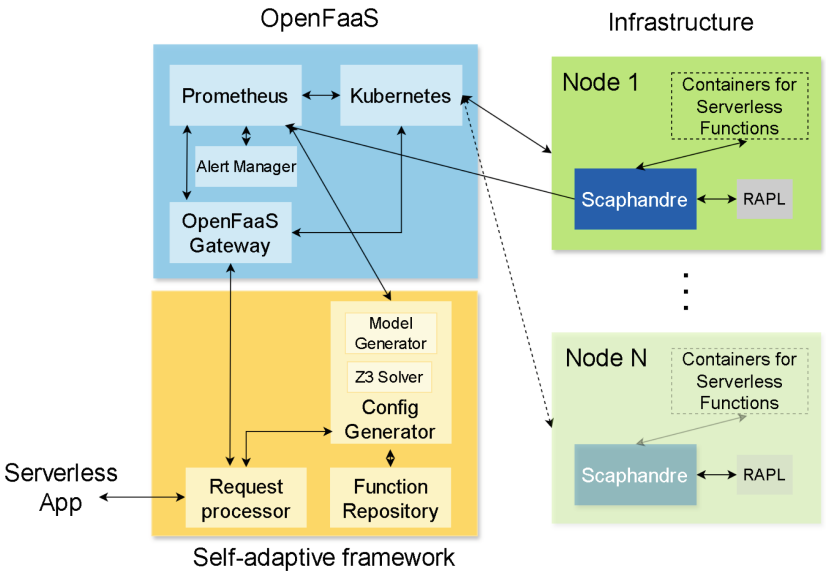

Fig. 5. Architecture of the whole system

The self-adaptive system (bottom left of Fig. 5) analyses the application based on Software Product Lines [7]. It uses information stored in a repository about the application's functions to generate automatically a set of Feature Models [7] that describe the variability of the application. Then, a solver calculates the optimal configuration that allows the system to achieve certain quality of service requirements. In the case of energy, we can combine it with the user experience so that the system chooses the functions that consume the least energy while maintaining a certain level of user experience. When the application requests a function, the system will use the selected configuration to decide which implementation of those available in the repository is actually executed.

The system can also be reconfigured at runtime to work self-adaptively with support for energy measurements, so, we need to add the appropriate rules to achieve it. To avoid overloading the system by continuously measuring the consumption of each function that makes up the serverless application, we have chosen to read this data only after each execution of each function. The data is processed as explained in the previous section, and the values obtained will

be used to compare them with those stored at that moment for that function. Initially, the repository may contain information about the values measured in previous executions or tests of the functions to compare the consumption of the different implementations during the first optimisation process, but this really would not be necessary. since the system updates this information at runtime.

The stored energy values are not updated each time a new measurement is made, but only when these measurements exceed or fall below a certain threshold, calculated as a percentage of that function's currently stored energy value. It is necessary to use thresholds with sufficient margins to determine whether a function has increased or decreased its consumption and whether the variation is due to measurements. A single one could be considered for the entire system, but, as we have verified that the functions can present significant differences in their value range, it is more appropriate to use a different one for each function. To calculate these variable thresholds, we use the same periods as for moving averages, we calculate the average, the maximum and minimum values, and we find the maximum percentage of variation between the average value and said extremes. We consider the greatest of these two results and set the threshold, adding a certain margin. Every time we update the energy values associated with a function, we will also update the considered threshold for that function.

As the period for moving averages, we can choose the one that best suits our needs. The higher it is, the more stable the energy measurements are, and we can use a tighter variation threshold, but it will take longer to react to changes in energy consumption. For example, we can choose 5 min for each moving average, so we can adjust thresholds of 15% for the worst case (*f6*) and 5% for the best (*f1*), which is quite acceptable. However, the times used to calculate the second moving average do not have to coincide with those of the first, so variations can also be made in that sense.

6 Case Study

We have developed a case study to verify that the system works in a self-adaptive way using the data provided by Scaphandre and the rules we have described to launch the reconfiguration process. As our objective is not to test the operation of the framework, we have chosen a simple case in which only one function varies, so that it is easier to observe the self-adaptation process. It can be extrapolated to a more complex system since the operation of the framework with other systems has been previously tested [10]. The application consists of a system for vehicle control. This controls the passage, and, for security reasons, it takes a photograph of the license plate and another of the driver, which must be stored for some time. The BPMN 2.0 diagram of the case study is shown in Fig. 6. It consists of a step control stage, another for taking photographs and another for storing pictures in a compressed file. To verify the system's response, we will focus on the compression function, for which two alternative implementations with different energy consumption are deployed, as shown in Fig. 7.

The tools used in our experiments are Scaphandre v1.0.0 and OpenFaaS v0.27. Our software is programmed using Python 3, and the hardware used is

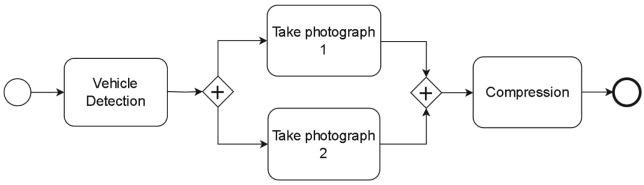

Fig. 6. BPMN 2.0 diagram of the sample application

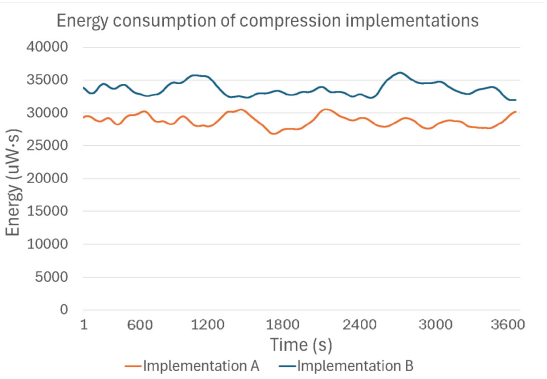

Fig. 7. Energy data obtained for compression functions

a PC Intel i5-7400, 3.00 GHz, 24 GiB of RAM. As an interval for the moving averages, we have chosen 5 min and adjusted the thresholds automatically for the same period, as explained in the previous section. This assures us that not only is the variability due to the measurements taken into account but also due to the data. As the photographs are of the same size and to similar objects, there is not expected to be much variation due to this. If there were, the system could also be used, but it would only make sense for long sampling periods so that the consumption data could be homogenised and thus be used to compare with that of another function.

To ensure that the measurements are stable, it will be necessary to wait 5 min for the first moving average to be correct and 5 more for the double to be correct. That is, we establish a setup time of 10 min, until which we will not start any reconfiguration process. When running the test application without starting information about the energy consumption of the functions, the chosen compression function can be any of the two available. We choose the worst case, which is that the application starts running the one with the highest consumption. We observe how, after the setup time has elapsed, the self-adaptation process is launched and the system chooses the alternative implementation, reducing the energy consumption of the application, as shown in Fig. 8 - Case A. For this case, in which 10 concurrent calls are made continuously for 1 h, the energy consumption due to functions compared to the estimated consumption that would have

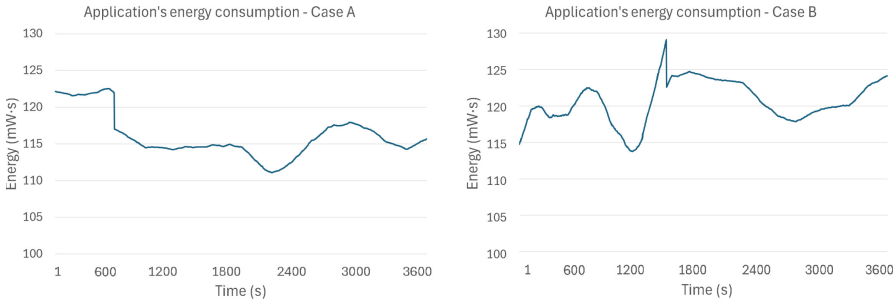

Fig. 8. Execution results for cases A and B

been generated if the reconfiguration was not performed, is 5%. It must be taken into account that the difference in consumption of the two implementations considered is not very high, as can be seen in Fig. 7 and now we are considering the whole application. On the other hand, in an application with more functions that present alternatives, the reduction may be greater.

We also reproduce a situation in which the energy consumption of the function increases. The system launches the reconfiguration process after exceeding the threshold previously calculated by the system, which in this case is 9.1% for that function. At that point, it starts running the alternative implementation, and the application's energy consumption is reduced, as can be observed in Fig. 8 - Case B. In this situation the reduction of consumption is about 2%. This reduction is lower because reconfiguration is done in minute 24 and the alternative implementation is the one that previously presented more consumption. However, it is important to keep in mind that these results are absolutely dependent on the specific application and the infrastructure on which it is deployed.

7 Conclusions

We have integrated energy measurement software into a FaaS environment to measure the energy consumed by serverless functions. Due to the high level of variation of the data obtained, some solutions have been proposed to be able to properly compare the energy consumption of various functions at runtime, presenting the improvements achieved. It has also been proven that the variations in the measured values differ depending on the type of function executed. Due to this, the use of variable thresholds has been proposed and a method to calculate them dynamically at runtime for each function has been presented. We have verified that the system presented works appropriately by implementing a case study of a self-adaptive system.

The effect of the infrastructure over the measures is challenging, so, as a line of future work, we plan to add information about the infrastructure on which the functions are executed to the analysis, to make better self-adaptation decisions.

Acknowledgmnts. Work supported by the *TASOVA PLUS* research network (RED-2022-134337-T), and the projects *IRIS* PID2021-122812OB-I00 (FEDER funds), *DAEMON* H2020-101017109.

References

1. Alhindi, A., Djemame, K., Heravan, F.B.: On the power consumption of serverless functions: An evaluation of openfaas. In: IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC), pp. 366–371. IEEE (2022)
2. Byrne, A., Pang, Y., Zou, A., Nadgowda, S., Coskun, A.K.: Microfaas: energy-efficient serverless on bare-metal single-board computers. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 754–759 (2022)
3. Ewim, D.R.E., Ninduwezuor-Ehiobu, N., Orikpete, O.F., Egbokhaebho, B.A., Fawole, A.A., Onunka, C.: Impact of data centers on climate change: a review of energy efficient strategies. J. Eng. Exact Sci. (2023)
4. Intel: RAPL (2012). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
5. Jia, X., Zhao, L.: Raef: Energy-efficient resource allocation through energy fungibility in serverless. In: IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS), pp. 434–441. IEEE (2021)
6. Le, D.N., Pal, S., Pattnaik, P.K.: OpenFaaS, chap. 17, pp. 287–303. John Wiley & Sons, Ltd (2022). https://doi.org/10.1002/9781119682318.ch17
7. Lee, K., Kang, K.C., Lee, J.: Concepts and guidelines of feature modeling for product line software engineering. In: Gacek, C. (ed.) Software Reuse: Methods Techniques Tools, pp. 62–77. Springer, Berlin Heidelberg (2002)
8. Manotas, I., et al.: An empirical study of practitioners' perspectives on green software engineering. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, pp. 237-248. ACM, New York (2016)
9. Moreno-Vozmediano, R., Huedo, E., Montero, R.S., Llorente, I.M.: Latency and resource consumption analysis for serverless edge analytics. J. Cloud Comput. **12**(1), 108 (2023). https://doi.org/10.1186/s13677-023-00485-9
10. Serrano-Gutierrez, P., Ayala, I., Fuentes, L.: Fuspaq: a function selection platform to adjust qos in a faas application. In: Service-Oriented Computing – ICSOC 2022 Workshops, pp. 249–260. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-26507-5_20
11. Siddik, M.A.B., Shehabi, A., Marston, L.: The environmental footprint of data centers in the United States. Environ. Res. Lett. **16**(6), 064017 (2021). https://doi.org/10.1088/1748-9326/abfba1
12. Tsenos, M., Peri, A., Kalogeraki, V.: Energy efficient scheduling for serverless systems. In: 2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), pp. 27–36 (2023)
13. Tzenetopoulos, A., Marantos, C., Gavrielides, G., Xydis, S., Soudris, D.: Fade: Faas-inspired application decomposition and energy-aware function placement on the edge. In: Proceedings of the 24th International Workshop on Software and Compilers for Embedded Systems, SCOPES 2021, pp. 7-10. Association for Computing Machinery, New York (2021)https://doi.org/10.1145/3493229.3493306

Code Reuse

Using Code from ChatGPT: Finding Patterns in the Developers' Interaction with ChatGPT

Anastasia Terzi(✉), Stamatia Bibi, Nikolaos Tsitsimiklis, and Pantelis Angelidis

University of Western Macedonia, Kozani, Greece
{a.terzi,sbibi,ece01243,paggelidis}@uowm.gr

Abstract. ChatGPT can advise developers and provide code on how to fix bugs, add new features, refactor, reuse, and secure their code but currently, there is little knowledge about whether the developers trust ChatGPT's responses and actually use the provided code. In this context, this study aims to identify patterns that describe the interaction of developers with ChatGPT with respect to the characteristics of the prompts and the actual use of the provided code by the developer. We performed a case study on 267,098 lines of code provided by ChatGPT related to commits, pull requests, files of code, and discussions between ChatGPT and developers. Our findings show that developers are more likely to integrate the given code snapshot in their code base when they have provided information to ChatGPT through several rounds of brief prompts that include problem-related specific words instead of using large textual or code prompts. Results also highlight the ability of ChatGPT to handle efficiently different types of problems across different programming languages.

Keywords: ChatGPT · bugs · features · reusing · developer interaction · association rules · case study · prompts · pattern concepts

1 Introduction

The need for the timely release of defect-free software along with the accelerated development cycles places great pressure on software developers who often seek quick, high-quality solutions to the programming challenges that they confront [20]. The rise of ChatGPT introduced a new dimension to the developer community's quest for answers and solutions by accumulating and synthesizing knowledge from various sources, offering instantaneous and personalized assistance.

Currently, ChatGPT has emerged as a game-changer in the developer landscape, leveraging the power of Natural Language Processing to generate text that aids in multi-purpose tasks. Even though it was not primarily introduced for software engineering, it seems to be able to provide advice on how to fix

A. Achilleos et al. (Eds.): ICSR 2024, LNCS 14614, pp. 137–152, 2024.
https://doi.org/10.1007/978-3-031-66459-5_9

bugs, add new features, improve already-existent code, or even suggest tools and libraries to be reused for a specific purpose [13]. ChatGPT undoubtedly has disrupted the traditional paradigm of seeking programming help, mainly due to its judgment-free environment that provides fast responses. However, the question that arises is how relevant and accurate these responses are and whether developers trust and utilize the proposed code.

Up to now, there is little knowledge about how the developers use ChatGPT and whether the responses provided are relevant. This study attempts to shed light on the primary concerns raised by software developers in ChatGPT conversations, with respect to code acquisition, and the context under which the code provided is actually used. This involves analyzing the nature of concerns (programming language prompt-ed, tokens provided) and potential challenges (types of development tasks prompted), along with the developers' code base (GitHub repositories). The goal of the study is to identify frequently appearing patterns, in the form of association rules, that de-scribe the correlation between prompt characteristics and the utilization (or not) of the generated code snippet. On this end, the study pursues two key goals:

- The first goal is to identify the main concerns of the developers when interacting with ChatGPT. For this purpose, we identify patterns that reflect the main issues presented by developers to ChatGPT along with the associated programming languages, and the type of development task in which they seek assistance. Through this first goal, the research will provide insights into common prompting patterns that reflect the needs of the developers' community that are expected to be ad-dressed by ChatGPT.
- The second goal explores the various ways developers engage with ChatGPT and how these interaction styles influence the use (or not) of code from generated responses. The analysis will highlight whether developers primarily provide detailed instructions or adopt a more conversational approach. Furthermore, the research investigates whether a correlation exists between interaction styles and code utilization, offering recommendations on how developers should tailor their interactions to maximize the effectiveness of ChatGPT's contribution.

To address these inquiries, we conducted a case study utilizing DevGPT, a dataset comprising developer-ChatGPT conversations, including prompts and responses encompassing code snippets and associated GitHub repositories. Specifically, our analysis focused only on conversations where ChatGPT provided source code. We analyzed conversations concerning GitHub commits, GitHub pull requests, GitHub discussions, raw code files, and Hacker news posts. As a next step, we examined whether the code provided by ChatGPT was actually used by the developer, by checking the similarity of the provided code snippet in the project's GitHub repository. Then we searched for frequently appearing patterns in this context, aiming to identify the con-text under which developers use ChatGPT's generated code. The rest of the paper is organized as follows: Sect. 2 summarizes the key contributions of previous research related to our topic

of interest. Section 3 outlines the design of our case study, including the methodologies of data collection and analysis. Section 4 presents the results, organized by research question while in Sect. 5, we provide a discussion on the meaning and significance of the findings, we also address any potential limitations or threats to the validity of our study. Finally, in Sect. 6, we conclude the paper.

2 Related Work

Large-scale language models (LLMs) [6]are rapidly being adopted by software developers and applied to generate code and other artifacts associated with software engineering. Popular examples of LLM-based tools applied for these purposes include ChatGPT [15] and GitHub Copilot [10]. Discussions on LLM adoption in software engineering processes have centered on automated code generation and the security and code quality risks associated with the generated code. For example, Asare et al. [3] compared LLM code generation to humans from a security perspective. Similar research [4]has examined the quality of LLM-generated answers and code and LLM interaction patterns for fixing bugs [11]. The majority of studies found in the literature focus mainly on the ability of LLMS to fix programming bugs or focus on the security aspect of produced code. Surameery and Shakor [17] highlighted the potential of ChatGPT as a comprehensive debugging toolkit but stressed the need to be used supplementary with other debugging tools. Sobania et al. [16] used a range of bug scenarios to test ChatGPT's efficacy in fixing bugs and concluded that it is competitive with the other automated tools while the interactive dialog window offered by ChatGPT increases the bug fix success rate. [16]

In a similar direction, Jalil et al. [7], examined in an educational environment, the replies of ChatGPT in BUG FIX scenarios and found that 44% of them are correct or partially correct, while the explanations provided are correct or partially correct in 57% of the cases. Other studies focused on ChatGPT's efficiency in providing answers to software architecture problems [1] and coding for various domains such as: in the field of 3D printing [12], in creating new general intelligence bots [9], and in typical numerical problems [8].

In all these studies the main conclusion is that ChatGPT when fed with suitable prompts can generate in several cases the relevant code leading to significant time saving. Furthermore, two studies suggest the use of prompt patterns to increase the probability of receiving an appropriate answer from ChatGPT [18]. White et al., in [18] used a catalog of fast patterns and concluded that the proposed catalog improves by 4% the performance of ChatGPT. Similarly, in [2] the authors suggested a set of prompt patterns for tasks related to code quality improvement, refactoring, requirements elicitation, and software design. Our research draws inspiration from these explorations and documents specific pat-terns that can be used to generate the desired code outcome from LLM models. The main contribution of this study is that, to our knowledge, this is the first study that examines the use of ChatGPT, and the appropriateness of the provided responses, in real-life multi-purpose code prompts that are associated with pre-existing projects hosted in GitHub. The access to the associated

GitHub repository gives us the chance to study the level of originality of ChatGPT's results and verify whether the code included in the answer of ChatGPT was actually reused by practitioners and to which extent.

3 Case Study Design

In this section, we present the study design, based on the guidelines of Runeson et al. [14]

3.1 Research Questions

As discussed in the introductory section, the main objective of the present study is to explore the capability of ChatGPT to useful code and to evaluate the level of trust developers place in the utilization of the generated code. To align our research methodology with the underlying motivations, we define the following research questions and objectives:

RQ1: What are the main concerns of the developers when interacting with ChatGPT?

This research question aims to identify the main issues that developers present to ChatGPT and to provide an overall overview considering their frequency, characteristics, and potential repercussions. In order to do this, we analyze the prompts that developers provide to ChatGPT in an attempt to deliver an overview that includes their frequency, characteristics, and possible outcomes. We isolate repetitive keywords from the titles, that are used to describe the prompts and identify patterns that are indicative of the co-existence of certain tasks. The analysis provides six different categories of tasks, which are: *BUG FIX, ENERGY, NEW FEATURE, REFACTOR, SECURITY*, and *OTHER* tasks. With this classification, we want to conclude whether there are dominant types of requests when interacting with LLMs such as ChatGPT.

RQ2: Are there any patterns that describe the interaction of developers with ChatGPT that are associated with the actual use of the response?

The purpose of RQ2 is to investigate the factors that influence the adoption of code generated by ChatGPT. Focusing on prompts, questions, and inherent characteristics of the interaction process we are looking for patterns that help towards receiving a useful answer from ChatGPT. For this reason, we first identify the originality of the AI-generated code compared to the given input. Sections of the code that resemble the original query submitted to ChatGPT are filtered out. In this question, we check whether developers are relying on ChatGPT's answers. For this reason, we examine if the after filtered code provided as an answer by ChatGPT can be found unaltered in the project repository mentioned by the developer. This question aims to highlight developers' reliance on LLMs.

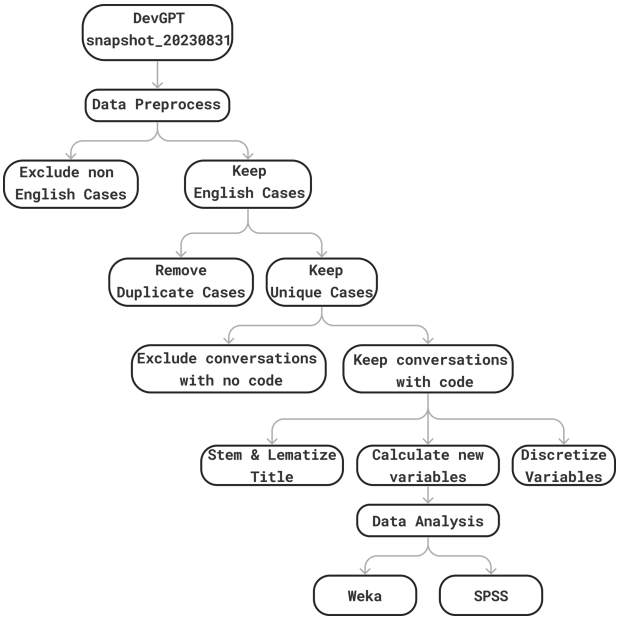

Fig. 1. Study map.

To answer the aforementioned questions, the process presented in Fig. 1 was applied. The outlined approach consists of four phases that are (i) data collection, (ii) data preprocessing, (iii) discretization of variables with continuous values, (iv) data analysis, and (v) extraction of results accompanied by discussion of the most important findings.

3.2 Data Collection

The dataset utilized in this study is DevGPT[1], a curated dataset to explore how software developers interact with ChatGPT. The dataset is derived from shared ChatGPT conversations collected from GitHub and Hacker News. The moment this study took place, November 2023, the dataset encompassed 17,913 prompts and responses from ChatGPT, including 11,751 code snippets, that are linked to corresponding software development artifacts such-ranging from source code, commits, issues, pull requests, to discussions and Hacker News threads. The under-study snapshot, snapshot20230831[2], consists of a total of 2,714 conversations between ChatGPT and users.

The DevGPT dataset is launched as a collection of six sets of JSON files. Each file includes the URL to the ChatGPT conversation[3], associated HTTP response

[1] https://github.com/NAIST-SE/DevGPT.
[2] https://github.com/NAIST-SE/DevGPT/tree/main/snapshot_20230831.
[3] https://help.openai.com/en/articles/7925741-chatgpt-shared-links-faq.

status codes, access date of the URL, and the HTML response content and a link to the developer's GitHub repository. Additionally, each conversation contains a list of prompts/answers, inclusive of any code snippets. The dataset provides details including the date of the conversation, the count of prompts/answers, their token information, and the model version involved in the chat.

From the enclosed information we kept the data that are related to: prompt title, number of prompts, the amount of used prompt (*NumberOfPrompts*) the number of tokens provided by ChatGPT's answers (*TokensOfAnswers*), the code snippets given by ChatGPT along with the number of the lines of code (*Lines-GPT*) and links to the associated GitHub repositories, the programming language of the repository, the lines used by the developer in the aforementioned repository (*RepoLanguage*). We analyzed in total 2399 conversations out of which 379 commits, 138 pull requests, 347 issues, 40 discussions, 1305 code files, and 190 hacker news. The rest 315 cases were excluded.

3.3 Data Pre-processing

The next step of the methodology involved the detection of the most useful fields to be analyzed, in accordance with the goals and objectives set by the proposed RQs.

Initially, in the Data Pre-Processing part of our Analysis we implemented a script to handle the raw JSON files contained within the snapshot under investigation. and then we reviewed the processed data manually to identify and rectify any potential errors or discrepancies. As a next step we entered the data to Weka to check their reliability and further analyze its insights. The data filtering process, we followed a three-step process:

(a) We excluded all the cases where the language of user's prompt was different than English since keeping those cases would add uncertainty to our produced results.

(b) We removed all duplicate conversations. To determine a duplicate conversation, we checked all the attributes of the interaction including a) Link of conversation, b) Link of repository c) Individual prompts.

(c) We excluded all conversations where the Generative AI model did not include any code snippet in its answer.

After the filtering process the remain Dataset consisted of 2399 unique interactions. Regarding the data preparation the process included:

(a) Identification of tasks related to the prompts (Type): For this purpose, we extracted the separate words from the title of each prompt, with stemming techniques we removed associative and irrelevant words (and it, how, please, etc.) and through lemmatization we grouped words with the same root (i.e. add-> addition). As a second step we proceeded by classifying together words of similar context in six task categories using a keyword-based sub-string search method adopted from [5]. Additionally, we introduced a sixth category labeled *OTHER* to describe the cases that did not fall into any of the five categories yet led to code utilization.

In the end, each prompt was characterized as one of the following tasks: *BUG FIX, ENERGY awareness, NEW FEATURE, REFACTOR, SECURITY, and OTHER* software-related tasks.

(b) Calculation of the unique code lines provided by ChatGPT (*LinesGivenGPT*): This variable indicates the number of unique lines of code provided by ChatGPT that the developer used. That were not included in the initial prompt of the developer. To calculate the value of the variable we first checked the code provided by the developer and compared this code with the code snippet provided by ChatGPT. Then we kept only the unique lines provided by ChatGPT (*LinesGivenGPT*)

(c) Calculation of the percentage of code lines, provided by ChatGPT, that were used by the developer (*LinesUsed*): To calculate this variable we searched for the unique code lines provided by ChatGPT (*LinesGivenGPT*) within the GitHub repository linked to the associated conversation. In the case where part of the code was used, the additional variable *LinesUsed* was calculated as the number of lines of code provided by ChatGPT that were used by the developer to the total number of unique lines provided by ChatGPT. The variables that participated in the study are presented in Table 1 along with a description. All variables were extracted from the given DevGPT dataset and were later accordingly reformed though manual and automated processes.

Table 1. Variables Overview.

Variable	Description
Type	Extracted type of JSON file (*FILE CODE, COMMIT, ISSUE, PULL REQUEST, DISCUSSION, and HACKER NEWS THREADS*)
RepoLanguage	The main language of the referred repository. For the purpose of our study we focused on (*HTML, PYTHON, JS, GO, C, BASH, JAVA*)
Type	Related to the question context (*BUG FIX, ENERGY, NEW FEATURE, REFACTOR, SECURITY, and OTHER*)
NumberOfPrompts	Number of prompts in this conversation
TokensOfPrompts	Number of tokens of prompts in this conversation
TokensOfAnswers	Tokens of answers in this conversation
Used CODE	Binary variable that states whether developer provided code
LOC	Number of code lines provided by the developer
CodeGPT	The main language of the answer assigned by ChatGPT
LinesGPT	Code lines provided by ChatGPT
LinesGivenGPT	Unique code lines provided by ChatGPT, after excluding code lines given by the user
LinesUsed	Percentage of unique code lines provided by ChatGPT that were used by the developer

3.4 Discretization of Variables with Continuous Values

Discretization of variables is used in data processing and involves partitioning the data range into intervals and assigning data points to their corresponding interval or categories. Discretization is essential for data mining processes as it facilitates the discovery of patterns, associations, and correlations. Equal frequency binning was chosen for this purpose, as it offers clear insights into the distribution of observations and can handle outliers effectively [19]. The ranges defining the classes used for discretizing continuous variables are detailed in Sect. 3. This pre-processing step ensures the reliability of the association rule mining process, enabling meaningful in-sights to be derived from the data.

Based on the original range of variables, the following Table 2 presents the 5 variables that were discretized along with the 5 distinct associated categories.

Table 2. Variable discretization in five classes.

Variable	Very Low	Low	Average	High	Very High
NumberOfPrompts	≤ 2	3	4–8	9–20	21+
TokensOfPrompts	≤ 241	242–685	686–1094	1095–3270	3271+
TokensOfAnswers	≤ 7564	757–1584	1585–3745	3746–6505	6506+
LinesUsed	0	1–3	4–8	9–25	26+
LinesGivenGPT	≤ 2	2–8	9–21	22–39	40+

3.5 Data Analysis

The data analysis for this case study consists of three main sections. First, we estimate the frequency and descriptive statistics of the prompt characteristics. This involves identifying the frequency that specific features appear along with providing statistical summaries to better understand the general distribution of prompts the results Then we implemented statistical analysis to determine if there is a distinction in the adoption of the ChatGPT code depending on several prompt features. This test assists with identifying potential links between prompt features and generated code use patterns [19].

After that, we isolated frequently appearing item sets via visualization graphs and Association Rules (ARs). Association Rules are part of descriptive modeling strategies that seek to explain data and underlying relationships. This is achieved by establishing a set of rules that collectively define the variables of interest, shedding light on the associations and patterns within the dataset. Through these analyses, we get insights into the dynamics of prompt characteristics and the subsequent code adoption.

Given a set of observations over attributes A1, A2, ..., An in a data set D a simple association rule has the following form:

$$A1 = X and A2 = Y => A3 = Z \tag{1}$$

$$Confidence = P(A3 = Z|A1 = X, A2 = Y) \quad (2)$$

$$Support = freq(X \cap Y \cap Z, D). \quad (3)$$

This rule is interpreted as follows: when attribute A1 has the value X and attribute A2 has the value Y then there is a probability p (Confidence) that attribute A3 has the value Z. Confidence (C) is the probability p defined as the percentage of the records containing X, Y, and Z regarding the overall number of records containing X and Y only. Support (S) is a measure that expresses the frequency of the rule and is the ratio between the number of records that present X, Y, and Z to the total number of records in the data set (D). In this study, we employed the Apriori algorithm to extract the Association Rules (ARs), with Weka.

4 Results

In this Section we present the results of this case study, organized by research question.

RQ1: What are the main concerns of the developers when interacting with ChatGPT?

To address the research question (RQ1) regarding the main concerns of developers when interacting with ChatGPT, we examined the distribution of conversation characteristics across different programming languages and types of conversation. Table 3 provides insights into the distribution of characteristics such as the number of prompts, tokens of prompts, and lines used across various programming languages. Similarly, Table 4 presents the distribution of conversation characteristics for different types of conversation, including *COMMIT, DISCUSSION, ISSUE, PULLREQUEST, HACKER NEWS, and CODE FILE.*

Table 3. Distribution of conversation characteristics regarding the Programming Language.

Prompt	BASH	C	HTML	JAVA	JS	NOPL	PYTHON	RESTPL
Number	6.86	9.72	9.47	5.81	4.97	4.92	6.30	6.73
Tokens	608.23	1594	1028	2383.31	635.61	700.97	1388.20	1091.45

The distribution of conversation characteristics in Table 3 and Table 4 indicates that languages like *JAVA and JAVASCRIPT*, which exhibit higher percentages of *BUG FIX* tasks, also tend to have higher numbers of prompts and tokens, indicating potentially more complex or extensive interactions in these languages. While the rest of the programming languages (*RESTPL*) show lower percentages of *BUG FIX* tasks alongside lower numbers of prompts and tokens, suggesting a different usage pattern or focus.

Table 4. Distribution of conversation characteristics regarding the type of conversation.

Prompt	COMMIT	DISCUSSION	ISSUE	PULL REQUEST	HACKER NEWS	CODE FILE
Number	2.98	3.87	4.01	5.67	6.02	9.92
Tokens	827.94	523.93	722.32	1736.03	708.46	1198.67

Analyzing Table 5, which depicts the distribution of tasks across different programming languages, we observe that tasks related to *REFACTOR* are prevalent across several languages, with the highest percentage observed in *BASH* and *HTML*. The *BUG FIX* task exhibits higher percentages in languages like *JAVA and JAVASCRIPT*.

Table 5. Distribution of conversation characteristics regarding the type of conversation.

	BASH	C	HTML	JAVA	JS	NOPL	PYTHON	RESTPL
SECURITY	2%	4%	41%	3%	**14%**	10%	19%	7%
BUG FIX	4%	11%	34%	**4%**	11%	14%	16%	5%
NEW FEATURE	2%	6%	33%	**4%**	**14%**	15%	**21%**	4%
REFACTOR	**5%**	8%	**53%**	2%	6%	9%	12%	5%
ENERGY	**5%**	10%	29%	0%	10%	**24%**	19%	5%
OTHER	3%	**12%**	32%	3%	12%	13%	17%	**9%**

In Table 6 we examine the interconnected nature of developer concerns when inter-acting with ChatGPT.

In particular Table 6 presents the occurrence of each type of task (% column), the percentage of the cases where the code provided by ChatGPT was utilized per each type of task (%Used column), and the pairwise frequency of tasks presented together in the same prompt.

REFACTOR tasks constitute the highest percentage of prompts (38.4%), indicating a prevalent need for code optimization and reuse among developers. Additionally, *BUG FIX* and *NEW FEATURE* implementation tasks are common, with moderate levels of utilization. *SECURITY*-related tasks, although less frequent, exhibit a relatively high utilization percentage, emphasizing their importance in ensuring the security of software systems. Lastly, a significant portion of prompts falls under the *OTHER* category, indicating a diverse range of developer needs beyond the primary task categories identified. The most frequent pair of tasks identified in the analysis is *NEW FEATURE-REFACTOR*. This combination appears in 47% of the prompts and exhibits a utilization rate of 23%. This suggests that users seem to initiate their prompts by pinpointing

specific programming tasks that require modification, due to new feature added bugs.

Table 6. Types of tasks prompted and the combinations of which led to code integration.

Task	%	%Used	SECURITY	BUG FIX	N.FEATURE	REFACTOR	ENERGY
SECURITY	5.1%	49.5%					
BUG FIX	20.6%	57.2%	2.8%				
N. FEATURE	32.5%	57.2%	4.8%	2.2%			
REFACTOR	**38.4%**	**53.8%**	**4.7%**	**12.2%**	**23%**		
ENERGY	0.9%	66%	9%	57%	61%	38%	
OTHER	27.7%	53.2%	7.8%	1.8%	8.2%	2%	0

Besides coupling keywords several cases included a combination with three or more keywords, making the prompts addressed to ChatGPT more specific. Among these the *NEW FEATURE REFACTOR BUG-FIX* is the most common triplet appearing in the dataset with 41% of the time resulting in code adoption. As presented in Table 7 when developers frequently seek advice from ChatGPT regarding security (*SECURITY*) concerns associated with newly added features (*NEW FEATURE*), often requesting guidance on refactoring (*REFACTOR*).

RQ1: Developers mostly prompt 'REFACTOR' tasks, with the first ones receiving an answer that is used in most of the cases ('53.8%'). 'REFACTOR' prompts are commonly combined with other tasks to address concerns and needs. Developers' concerns vary depending on the prompted language.

RQ2: Are there any patterns that describe the interaction of developers with ChatGPT that are associated with the actual use of the response?

In this question, we check whether the answer of ChatGPT (code snippet) is used or not by the developer along with the level of usage (*LinesUsed*). We first produced the association map of Fig. 2 which is a Relationship Graph between prompts' and answers' characteristics. The figure illustrates the connections between different combinations of values of the variables describing the

Table 7. Frequent task combination.

#	RULE	(S)	(C)
1	NEW FEATURE + SECURITY => REFACTOR	1.5%	63%
2	HTML => REFACTOR	43%	53%
3	NumberOfPrompts[VERY HIGH] => REFACTOR	16%	52%
4	PYTHON + REFACTOR => NEW FEATURE	4.7%	51%

Table 8. Association Rules leading to code utilization.

#	RULE	(S)	(C)
1	COMMIT + NumberOfPrompts[LOW] => LinesUsed[VERY HIGH]	15.6%	73%
2	CODE FILE + PYTHON + NumberOfPrompts[AVERAGE] + OTHER => LinesUsed[HIGH]	3.7%	71%
3	COMMIT + NumberOfPrompts[LOW] + NEW FEATURE => LinesUsed[VERY HIGH]	7.3%	70%
4	LOC [AVERAGE] + NumberofTokens [LOW] + BUG FIX => LinesUsed[AVERAGE]	4.5%	78%
5	LOC [LOW] + NumberofTokens [LOW] + SECURITY => LinesUsed[HIGH]	5.3%	78%
6	CODE FILE+ NEW FEATURE + JAVA => LinesUsed[HIGH]	1.2%	65%
7	LOC [AVERAGE] + NumberofTokens [LOW] + BUG FIX => LinesUsed[AVERAGE]	7%	94%
8	COMMIT + NumberOfPrompts[VERY LOW] + REFACTOR => LinesUsed[AVERAGE]	17.5%	60%
9	CODE FILE + NumberOfPrompts[LOW] + BUG FIX=> LinesUsed[AVERAGE]	5.3%	55%
10	CODE FILE + BUG FIX => LinesUsed[AVERAGE]	11%	51%
11	HACKER NEWS + NumberOfPrompts[VERY LOW] + BUG FIX => LinesUsed[LOW]	18.3%	89%
12	HACKER NEWS + BUG FIX => LinesUsed[LOW]	34%	65%
13	ISSUE + JAVASCRIPT + NEW FEATURE => LinesUsed[LOW]	9.8%	61%
14	CODE FILE + NEW FEATURE => LinesUsed[VERY LOW]	17.3%	63%
15	NO + NumberOfPrompts[VERY HIGH] => LinesUsed[NONE]	30%	92%
16	CODE FILE + REFACTOR => LinesUsed[NONE]	17.3%	84%
17	COMMIT +HTML => LinesUsed[NONE]	7.6%	74%
18	ISSUE + PYTHON => LinesUsed[NONE]	11%	50%
19	HTML+ NumberOfPrompts[LOW] + REFACTOR => LinesUsed[VERY HIGH]	9.8%	59%
20	JAVASCRIPT + NumberOfPrompts[AVERAGE] + NEW FEATURE => LinesUsed[VERY HIGH]	9.6%	53%
21	BASH + NumberOfPrompts[AVERAGE] + REFACTOR => LinesUsed[HIGH]	13%	77.3%
22	C + NumberOfPrompts[AVERAGE] => LinesUsed[HIGH]	16.8%	71%
23	HTML+ NumberOfPrompts[LOW] + REFACTOR => LinesUsed[VERY HIGH]	9.8%	59%
24	C + CODE FILE + REFACTOR => LinesUsed[HIGH]	22%	53%
25	HTML + NumberOfPrompts[VERY LOW] + REFACTOR => LinesUsed[AVERAGE]	16%	58%
26	C + NEW FEATURE => LinesUsed[AVERAGE]	22%	53%
27	HTML+ NumberOfPrompts[VERY LOW] + BUG FIX => LinesUsed[AVERAGE]	11.3%	53%
28	PYTHON + NEW FEATURE + REFACTOR=> LinesUsed[NONE]	9.8%	92%
29	C + BUG FIX => LinesUsed[NONE]	17%	75%
30	BASH + NumberOfPrompts[AVERAGE] => LinesUsed[NONE]	29%	62%
31	NumberofTokens [HIGH] + LOC[LOW] => LinesUsed[NONE]	17%	73%

prompts and the target variable, which is the *LinesUsed*. The edges in the model represent these connections, with their size indicating the frequency of the nodes and the thickness of the edges reflecting the strength of the association.

For instance, there is an association between the repository language HTML and the *Very Low* (≤ 2) number of prompts that is strong, i.e. presented in 201 instances. The same node led to *High and Very High* code adoption, hindering the need for specific keyword use rather than big explanatory texts to get the wanted result.

Additionally, to answer this RQ we identify frequently appearing patterns presented in Table 8.

The rule number 19 is interpreted as following: the use of several rounds of prompts highly focusing on refactoring issues when referring to an HTML task results in a Very High level of code adoption. Some general observations that stem from Table 8 are that interactions involving HTML or JAVASCRIPT combined with *REFACTOR* tasks tend to result in high code utilization. The

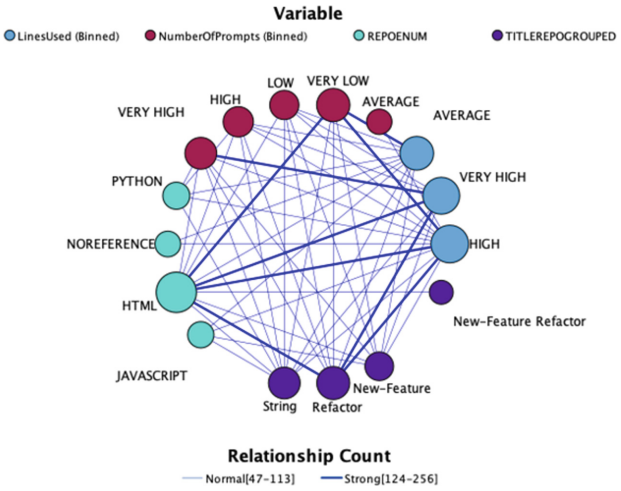

Fig. 2. Relationship Graph prompts', characteristics that led to code utilization.

characteristics of prompts significantly influence code adoption rates. Clarity (low/average number of prompts), specificity (code file provided), and relevance (mention a particular type of task for a specific programming language) contribute to higher levels of code utilization. A medium amount of back-and-forth interaction with GPT leads to better results than providing single long prompts or giving too much irrelevant info.

RQ2: Prompt characteristics influencing higher code adoption refer to specific tasks such as 'NEW FEATURE' and 'SECURITY' or interactions like 'COMMIT'. Developers should clearly state the general type of task and use sufficient rounds of prompts as input. Long textual prompts that are not accompanied by code is not a good practice, but so is long parts of code without the appropriate explanation.

5 Discussion

In this research, we observed that users often interacted with ChatGPT in an informal, conversational manner, even when seeking specific code-related assistance as in 47% of the cases the code provided by ChatGPT was not used. In particular, some observations that can be useful to practitioners are:

a. General textual input (high number of tokens) without being accompanied by technical information (code file, or adequate lines of code in the prompt) is in-sufficient. The informal interactions create a potential gap between user expectations and the system's response style.

b. Prompts related to small-scale programming tasks (*BUG FIX, SECURITY*) when accompanied with low or average lines of code are more likely to

receive code that it is going to be used. It seems that small tasks and context-specific prompts are more efficiently handled by ChatGPT.

c. Programming/Descriptive languages that are loosely coupled such as *JAVASCRIPT*, *HTML*, and shell languages (*BASH*) tend to be frequently prompted and receive answers that are going to be used. In particular, *REFACTOR* and *NEW FEATURE* prompts in these cases are efficiently addressed. d. Prompts that lack information in the text related to the Programming Language prompted or the specific type of task required also do not receive the code that is going to be used.

Based on these findings researchers can work on prompt engineering practices that are oriented on describing problems related to code generation. Prompt templates and good practices when prompting LLMs could increase the percentages of code utilization. Additionally, during our research, a large amount of ChatGPT's answer code was discarded as it was replicated or even copied from the same resource as the user's prompt. This raised a question on ethics, related to the source of ChatGPT-generated code, particularly concerning confidentiality and ownership matters. This issue should be researched and clarified prior to the formal integration of LLM's in the engineering lifecycle.

As a future work, we intend to explore how ChatGPT handles different types of is-sues or tasks specific to each programming language. By analyzing whether certain languages elicit more inquiries or if users consistently pose similar questions for each language, we can identify areas for improving model training. Moreover we are highly interested on investigating the resources and ethical considerations surrounding ChatGPT's code generation.

5.1 Threats to Validity

This section presents the threats to the validity of the current research formulated ac-cording to Runeson's et al. [14] classification.

Regarding Construct Validity, we should mention that possible bias could be inserted in the model by the variable related to the type of task in which the prompt was classified based on the keywords of the title, in the case where the classification mechanism changes then the results might be different. In the future, we intend to use and compare the results with different classification schema. Moreover, there is also bias in the calculation of the variable *UsedCODE* since we examine a particular GitHub repository time snapshot that differs across projects compared to the time when the prompt dialog took initially place. It is highly possible that the closer the dialog is to the snapshot we took the more likely is for the code to be used while as time passes the code after being tested might have been removed. The "survival" of ChatGPT code in the developers' repositories is also a subject of future research.

Internal validity, in this case, is not applicable since the examination of causal relationships is out of the scope of the study.

Concerning the **External validity** and in particular the generalizability supposition, changes in the responses might occur if the prompts analyzed were re-

fed to ChatGPT, a fact that would alter the results as well. A future replication of this study, on the same or different prompts would be valuable.

Additionally, DevGPTs sample size is limited. As a result, we encourage future studies to replicate our findings using bigger sample sizes and alternative datasets. As our research on this topic is ongoing, we plan to address the threats by expanding our sample size and exploring the hypothesis in a more diverse set of human-ChatGPT interactions.

To increase the **Reliability** of the study, that reflects the reproducibility of study, we applied two mitigation actions: (a) we recorded the case study design protocol in detail and (b) we uploaded the relevant tools that were used to obtain the data, along with the collected data in a GitHub repository.

6 Conclusions

ChatGPT offers developers guidance and code snippets for tasks like bug fixing, feature addition, and code security. The extent to which developers trust and utilize ChatGPT's responses remains unclear. Our study analyzed a dataset exceeding 267,098 lines of exchanged code. Interestingly, the research revealed that developers are more likely to integrate the provided code snippets when they use shorter, focused prompts that target specific problems over multiple interactions. Lengthy textual prompts or those that are already code-based seem to be less effective. Refactoring existing code or reusing existing solutions emerged as a prevalent task among developers, with a very high utilization rate (53.8%) for the provided responses from ChatGPT. Tasks addressing new features or security concerns, as well as interactions like commits, strongly influence the adoption of the provided code. To maximize effectiveness, our study suggests that developers should clearly articulate their tasks in concise prompts. Additionally, providing sufficient context and explanation along-side the code snippets for better understanding leads to better chances on code. Our study highlights the importance of tailored prompt strategies and clear communication in optimizing the utilization of ChatGPT's provided code. Our study represents the beginning of exploring the dependent relationship between large language models like ChatGPT and developers. Based on our findings, we aim to conduct further investigations to deepen our understanding of this relationship.

References

1. Ahmad, A., Waseem, M., Liang, P., Fahmideh, M., Aktar, M.S., Mikkonen, T.: Towards human-bot collaborative software architecting with ChatGPT. In: Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering (EASE '23), pp. 279–285. Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3593434.3593468
2. White, J., et al.: Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. arXiv (2023)

3. Asare, O., Nagappan, M., Asokan, N.: Is Github's copilot as bad as humans at introducing vulnerabilities in code? Empir. Softw. Engg. **28**(6) (2023). https://doi.org/10.1007/s10664-023-10380-1
4. Borji, A.: A categorical archive of ChatGPT failures. arXiv (2023)
5. Champa, A.I., Rabbi, M.F., Zibran, M.F., Islam, M.R.: Insights into female contributions in open-source projects. In: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), pp. 357–361. Melbourne, Australia (2023)
6. Bommasani, R., et al.: On the opportunities and risks of foundation models. arXiv (2021). https://doi.org/10.48550/arxiv.2108.07258
7. Jalil, S.E.A.: ChatGPT and software testing education: promises & perils. arXiv (2023)
8. Kashefi, M.T.: ChatGPT for programming numerical methods. J. Mach. Learn. Model. Comput. **4**(2), 1–74 (2023)
9. Kathikar, A., Nair, A., Lazarine, B., Sachdeva, A., Samtani, S.: Assessing the vulnerabilities of the open-source artificial intelligence (AI) landscape: a large-scale analysis of the hugging face platform. In: 2023 IEEE/ACM International Workshop on Automated Program Repair (APR) (2023). https://doi.org/10.1109/ISI58743.2023.10297271
10. Moradi Dakhel, A., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M.C., Jiang, Z.M.J.: Github copilot AI pair programmer: asset or liability? (2023). https://doi.org/10.1016/j.jss.2023.111734
11. Nasehi, S.M., Sillito, J., Maurer, F., Burns, C.: What makes a good code example?: A study of programming Q&A in stackoverflow. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM), pp. 25–34. Trento, Italy (2012). https://doi.org/10.1109/ICSM.2012.6405249
12. Owura, A., Nagappan, M., Asokan, N.: Is Github's copilot as bad as humans at introducing vulnerabilities in code? Empir. Softw. Eng. **28**(6) (2023). https://doi.org/10.1007/s10664-023-10380-1
13. Rahmaniar, W.: ChatGPT for software development: opportunities and challenges (2023)
14. Runeson, P., Höst, M., Rainer, A., Regnell, B.: Case Study Research in Software Engineering: Guidelines and Examples. Wiley, Hoboken (2012)
15. Silvia, B., Regondi, S., Frontoni, E., Pugliese, R.: Assessing the capabilities of ChatGPT to improve additive manufacturing troubleshooting. Adv. Ind. Eng. Poly. Res. **6**(3), 278–287 (2023). https://doi.org/10.1016/j.aiepr.2023.03.003
16. Sobania, M., Briesch, C., Hanna, J., Petke, J.: An analysis of the automatic bug fixing performance of ChatGPT. In: 2023 IEEE/ACM International Workshop on Automated Program Repair (APR), pp. 23–30. Melbourne, Australia (2023). https://doi.org/10.1109/APR59189.2023.00012
17. Surameery, N., Shakor, M.: Use ChatGPT to solve programming bugs. Int. J. Inf. Technol. Comput. Eng. **3**(31), 17–22 (2023)
18. White, J., et al.: A prompt pattern catalog to enhance prompt engineering with ChatGPT. arXiv (2023)
19. Witten, I., Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P.: The Weka data mining software: an update. SIGKDD Explor. **11**, 10–18 (2009). https://doi.org/10.1145/1656274.1656278
20. Xiao, T., et al.: 18 million links in commit messages: purpose, evolution, and decay (2023)

Evaluating the Reusability of Android Static Analysis Tools

Jean-Marie Mineau and Jean-Francois Lalande(✉)

CentraleSupélec, Inria, Univ Rennes, CNRS IRISA, Rennes, France
{jean-marie.mineau,jean-francois.lalande}@inria.fr

Abstract. Reproducibility and reusability in computer science experiments become a requirement for research works. Reproducibility ensures that results can be confirmed by using the same dataset and software of previous papers. Reusability helps other researchers to build new approaches with distributed software artifacts. For researchers in the field of security of mobile platforms, ensuring reproducibility and reusability is difficult to implement. In particular for reusability, datasets of Android applications may contain recent applications that past analysis software cannot process. As a consequence, past software produced by researchers may be difficult to reuse, which endangers the reproducibility of research. This paper intends to explore the reusability of past software dedicated to static analysis of Android applications. We pursue the community effort that identified publications between 2011 and 2017 that perform static analysis of mobile applications and we propose a method for evaluating the reusability of the associated tools. We extensively evaluate the success or failure of these tools on a dataset containing Android applications that can have up to six years of distance from the original publication. We also measure the influence of some important characteristics of the application such as being a goodware or a malware or the application size. Our results show that 54.5% of the evaluated tools are no longer usable and that the size of the bytecode and the min SDK version have the greatest influence on the reusability of tested tools.

1 Introduction

Android is the most used mobile operating system since 2014, and since 2017, it even surpasses Windows all platforms combined[1].The public adoption of Android is confirmed by application developers, with 1.3 millions apps available in the Google Play Store in 2014, and 3.5 millions apps available in 2017[2]. Its popularity makes Android a prime target for malware developers. Consequently, Android has also been an important subject for security research. In the past fifteen years, the research community released many tools to detect or analyze malicious behaviors in applications. Two main approaches can be distinguished: static and

[1] https://gs.statcounter.com/os-market-share#monthly-200901-202304.
[2] https://www.statista.com/statistics/266210.

A. Achilleos et al. (Eds.): ICSR 2024, LNCS 14614, pp. 153–170, 2024.
https://doi.org/10.1007/978-3-031-66459-5_10

dynamic analysis [18]. Dynamic analysis requires to run the application in a controlled environment to observe runtime values and/or interactions with the operating system. For example, an Android emulator with a patched kernel can capture these interactions but the modifications to apply are not a trivial task. As a consequence, a lot of efforts have been put in static approaches, which is the focus of this paper.

The usual goal of a static analysis is to compute data flows to detect potential information leaks [5,12,15,16,23,31,33] by analyzing the bytecode of an Android application. The associated developed tools should support the Dalvik bytecode format, the multiplicity of entry points, the event driven architecture of Android applications, the interleaving of native code and bytecode, possibly loaded dynamically, the use of reflection, to name a few. All these obstacles threaten the research efforts. When using a more recent version of Android or a recent set of applications, the results previously obtained may become outdated and the developed tools may not work correctly anymore.

In this paper, we study the reusability of open source static analysis tools that appeared between 2011 and 2017, on a recent Android dataset. The scope of our study is **not** to quantify if the output results are accurate for ensuring reproducibility, because all the studied static analysis tools have different goals in the end. On the contrary, we take as hypothesis that the provided tools compute the intended result but may crash or fail to compute a result due to the evolution of the internals of an Android application, raising unexpected bugs during an analysis. This paper intends to show that sharing the software artifacts of a paper may not be sufficient to ensure that the provided software would be reusable.

Thus, our contributions are the following. We carefully retrieved static analysis tools for Android applications that were selected by Li *et al.* [18] between 2011 and 2017. We contacted the authors, whenever possible, for selecting the best candidate versions and to confirm the good usage of the tools. We rebuild the tools in their original environment and we plan to share our Docker images with this paper. We evaluated the reusability of the tools by measuring the number of successful analysis of applications taken in a custom dataset that contains more recent applications (62 525 in total). The observation of the success or failure of these analysis enables us to answer the following research questions:

RQ1: What Android static analysis tools that are more than 5 years old are still available and can be reused without crashing with a reasonable effort?
RQ2: How the reusability of tools evolved over time, especially when analyzing applications that are more than 5 years far from the publication of the tool?
RQ3: Does the reusability of tools change when analyzing goodware compared to malware?

The paper is structured as follows. Section 2 presents a summary of previous works dedicated to Android static analysis tools. Section 3 presents the methodology employed to build our evaluation process and Sect. 4 gives the associated experimental results. Section 5 discusses the limitations of this work and gives some takeaways for future contributions. Section 6 concludes the paper.

2 Related Work

We review in this section the past existing datasets provided by the community and the papers related to static analysis tools reusability.

2.1 Application Datasets

Computing if an application contains a possible information flow is an example of a static analysis goal. Some datasets have been built especially for evaluating tools that are computing information flows inside Android applications. One of the first well known dataset is DroidBench, that was released with the tool Flowdroid [2]. Later, the dataset ICC-Bench was introduced with the tool Amandroid [33] to complement DroidBench by introducing applications using Inter-Component data flows. These datasets contain carefully crafted applications containing flows that the tools should be able to detect. These hand-crafted applications can also be used for testing purposes or to detect any regression when the software code evolves. Contrary to real world applications, the behavior of these hand-crafted applications is known in advance, thus providing the ground truth that the tools try to compute. However, these datasets are not representative of real-world applications [26] and the obtained results can be misleading.

Contrary to DroidBench and ICC-Bench, some approaches use real-world applications. Bosu *et al.* [5] use DIALDroid to perform a threat analysis of Inter-Application communication and published DIALDroid-Bench, an associated dataset. Similarly, Luo *et al.* released TaintBench [22] a real-world dataset and the associated recommendations to build such a dataset. These datasets confirmed that some tools such as Amandroid [33] and Flowdroid [2] are less efficient on real-world applications. These datasets are useful for carefully spotting missing taint flows, but contain only a few dozen of applications.

Pauck *et al.* [25] used those three datasets to compare Amandroid [33], DIALDroid [5], DidFail [15], DroidSafe [12], FlowDroid [2] and IccTA [16] – all these tools will be also compared in this paper. To perform their comparison, they introduced the AQL (Android App Analysis Query Language) format. AQL can be used as a common language to describe the computed taint flow as well as the expected result for the datasets. It is interesting to notice that all the tested tools timed out at least once on real-world applications, and that Amandroid [33], DidFail [15], DroidSafe [12], IccTA [16] and ApkCombiner [17] (a tool used to combine applications) all failed to run on applications built for Android API 26. These results suggest that a more thorough study of the link between application characteristics (e.g. date, size) should be conducted. Luo *et al.* [22] used the framework introduced by Pauck *et al.* to compare Amandroid [33] and Flowdroid [2] on DroidBench and their own dataset TaintBench, composed of real-world android malware. They found out that those tools have a low recall on real-world malware, and are thus over adapted to micro-datasets. Unfortunately, because AQL is only focused on taint flows, we cannot use it to evaluate tools performing more generic analysis.

2.2 Static Analysis Tools Reusability

Several papers have reviewed Android analysis tools produced by researchers. Li *et al.* [18] published a systematic literature review for Android static analysis before May 2015. They analyzed 92 publications and classified them by goal, method used to solve the problem and underlying technical solution for handling the bytecode when performing the static analysis. In particular, they listed 27 approaches with an open-source implementation available. Nevertheless, experiments to evaluate the reusability of the pointed out software were not performed. We believe that the effort of reviewing the literature for making a comprehensive overview of available approaches should be pushed further: an existing published approach with a software that cannot be used for technical reasons endanger both the reproducibility and reusability of research.

A first work about quantifying the reusability of static analysis tools was proposed by Reaves *et al.* [28]. Seven Android analysis tools (Amandroid [33], AppAudit [35], DroidSafe [12], Epicc [24], FlowDroid [2], MalloDroid [9] and TaintDroid [8]) were selected to check if they were still readily usable. For each tool, both the usability and results of the tool were evaluated by asking auditors to install and use it on DroidBench and 16 real world applications. The auditors reported that most of the tools require a significant amount of time to setup, often due to dependencies issues and operating system incompatibilities. Reaves *et al.* propose to solve these issues by distributing a Virtual Machine with a functional build of the tool in addition to the source code. Regrettably, these Virtual Machines were not made available, preventing future researchers to take advantage of the work done by the auditors. Reaves *et al.* also report that real world applications are more challenging to analyze, with tools having lower results, taking more time and memory to run, sometimes to the point of not being able to run the analysis. We will confirm and expand this result in this paper with a larger dataset than only 16 real-world applications.

3 Methodology

3.1 Collecting Tools

We collected the static analysis tools from [18], plus one additional paper encountered during our review of the state-of-the-art (DidFail [15]). They are listed in Table 1, with the original release date and associated paper. We intentionally limited the collected tools to the ones selected by Li *et al.* [18] for several reasons. First, not using recent tools enables to have a gap of at least 5 years between the publication and the more recent APK files, which enables to measure the reusability of previous contributions with a reasonable gap of time. Second, collecting new tools would require to describe these tools in depth, similarly to what have been performed by Li *et al.* [18], which is not the primary goal of this paper. Additionally, selection criteria such as the publication venue or number of citations would be necessary to select a subset of tools, which would require an additional methodology. These possible contributions are left for future work.

Table 1. Considered tools [18]: availability and usage reliability

Tool	Availability Bin	Src	Doc	Repo type	Decision	Comments
A3E [3] (2013)	–	●	●	github	×	Hybrid tool (static/dynamic)
A5 [32] (2014)	–	●	×	github	×	Hybrid tool (static/dynamic)
Adagio [10] (2013)	–	●	●	github	●	
Amandroid [33] (2014)	●	●	●	github	●	
Anadroid [19] (2013)	×	●	●	github	●	
Androguard [7] (2011)	–	●	●●	github	●	
Android-app-analysis [11] (2015)	×	●	●●	google	×	Hybrid tool (static/dynamic)
Apparecium [31] (2015)	●	●	×	github	●	
BlueSeal [30] (2014)	×	●	○	github	●	
Choi et al. [6] (2014)	×	●	○	github	×	Works on source files only
DIALDroid [5] (2017)	●	●	●	github	●	
DidFail [15] (2014)	●	●	○	bitbucket	●	
DroidSafe [12] (2015)	×	●	●	github	●	
Flowdroid [2] (2014)	●	●	●●	github	●	
Gator [29,36] (2014), (2015)	×	●	●●	edu	●	
IC3 [23] (2015)	●	●	○	github	●	
IccTA [16] (2015)	●	●	●	github	●	
Lotrack [20] (2014)	×	●	×	github	○	Authors ack. a partial doc.
MalloDroid [9] (2012)	–	●	●	github	●	
PerfChecker [21] (2014)	×	×	○	request	●	Binary obtained from authors
Poeplau *et al.* [27] (2014)	×	○	×	github	×	Related to Android hardening
Redexer [14] (2012)	×	●	●	github	●	
SAAF [13] (2013)	●	●	●	github	●	
StaDynA [37] (2015)	×	●	●	request	×	Hybrid tool (static/dynamic)
Thresher [4] (2013)	×	●	●	github	○	Not built with authors help
Wognsen *et al.* [34] (2014)	–	●	×	bitbucket	●	

binaries, sources: –: not relevant, ●: available, ○: partially available, ×: not provided
documentation: ●●: excellent, MWE, ●: few inconsistencies, ○: bad quality, ×: not available
decision: ●: considered; ○: considered but not built; ×: out of scope of the study

Some tools use hybrid analysis (both static and dynamic): A3E [3], A5 [32], Android-app-analysis [11], StaDynA [37]. They have been excluded from this paper. We manually searched the tool repository when the website mentioned in the paper is no longer available (e.g. when the repository have been migrated from Google code to GitHub for example) and for each tool we searched for:

- an optional binary version of the tool that would be usable as a fall back (if the sources cannot be compiled for any reason);
- the source code of the tool;
- the documentation for building and using the tool with a MWE (Minimum Working Example).

In Table 1 we rated the quality of these artifacts with "●" when available but may have inconsistencies, a "○" when too much inconsistencies (inaccurate remarks about the sources, dead links or missing parts) have been found, a "×" when no documentation have been found, and a double "●●" for the documentation when it covers all our expectations (building process, usage, MWE).

Results show that documentation is often missing or very poor (e.g. Lotrack), which makes the rebuild process very complex and the first analysis of a MWE.

Table 2. Selected tools, forks, selected commits and running environment

Tool	Origin		Alive Forks		Last commit	Authors	Environment
	Stars	Alive	Nb	Usable	Date	Reached	Language - OS
Adagio [10]	74	●	0	×	2022-11-17	●	Python - U20.04
Amandroid [33]	161	×	2	×	2021-11-10	●	Scala - U22.04
Anadroid [19]	10	×	0	×	2014-06-18	×	Scala/Java/Py. - U22.04
Androguard [7]	4430	●	3	×	2023-02-01	×	Python - Python 3.11 slim
Apparecium [31]	0	×	1	×	2014-11-07	×	Python - U22.04
BlueSeal [30]	0	×	0	×	2018-07-04	●	Java - U14.04
DIALDroid [5]	16	×	1	×	2018-04-17	×	Java - U18.04
DidFail [15]	4	×			2015-06-17	●	Java/Python - U12.04
DroidSafe [12]	92	×	3	×	2017-04-17	●	Java/Python - U14.04
Flowdroid [2]	868	●	1	×	2023-05-07	●	Java - U22.04
Gator [29,36]					2019-09-09	●	Java/Python - U22.04
IC3 [23]	32	×	3	●	2022-12-06	×	Java - U12.04 / 22.04
IccTA [16]	83	×	0	×	2016-02-21	●	Java - U22.04
Lotrack [20]	5	×	2	×	2017-05-11	●	Java - ?
MalloDroid [9]	64	×	10	×	2013-12-30	×	Python - U16.04
PerfChecker [21]		×			–	●	Java - U14.04
Redexer [14]	153	×	0	×	2021-05-20	●	Ocaml/Ruby - U22.04
SAAF [13]	35	×	5	×	2015-09-01	●	Java - U14.04
Thresher [4]	31	×	1	×	2014-10-25	●	Java - U14.04
Wognsen *et al.* [34]				×	2022-06-27	×	Python/Prolog - U22.04

●: yes, ×: no, UX.04: Ubuntu X.04

We finally excluded Choi *et al.* [6] as their tool works on the sources of Android applications, and Poeplau *et al.* [27] that focus on Android hardening. As a summary, in the end we have 20 tools to compare. Some specificities should be noted. The IC3 tool will be duplicated in our experiments because two versions are available: the original version of the authors and a fork used by other tools like IccTa. For Androguard, the default task consists of unpacking the bytecode, the resources, and the Manifest. Cross-references are also built between methods and classes. Because such a task is relatively simple to perform, we decided to duplicate this tool and ask to Androguard to decompile an APK and create a control flow graph of the code using its decompiler: DAD. We refer to this variant of usage as androguard_dad. For Thresher and Lotrack, because these tools cannot be built, we excluded them from experiments.

Finally, starting with 26 tools of Table 1, with the two variations of IC3 and Androguard, we have in total 22 static analysis tools to evaluate in which two tools cannot be built and will be considered as always failing.

3.2 Source Code Selection and Building Process

In a second step, we explored the best sources to be selected among the possible forks of a tool. We reported some indicators about the explored forks and our

decision about the selected one in Table 2. For each source code repository called "Origin", we reported in Table 2 the number of GitHub stars attributed by users and we mentioned if the project is still alive (• in column Alive when a commit exist in the last two years). Then, we analyzed the fork tree of the project. We searched recursively if any forked repository contains a more recent commit than the last one of the branch mentioned in the documentation of the original repository. If such a commit is found (number of such commits are reported in column Alive Forks Nb), we manually looked at the reasons behind this commit and considered if we should prefer this more up-to-date repository instead of the original one (column "Alive Forks Usable"). As reported in Table 2, we excluded all forks, except IC3 for which we selected the fork JordanSamhi/ic3, because they always contain experimental code with no guarantee of stability. For example, a fork of Aparecium contains a port for Windows 7 which does not suggest an improvement of the stability of the tool. For IC3, the fork seems promising: it has been updated to be usable on a recent operating system (Ubuntu 22.04 instead of Ubuntu 12.04 for the original version) and is used as a dependency by IccTa. We decided to keep these two versions of the tool (IC3 and IC3_fork) to compare their results.

Then, we self-allocated a maximum of four days for each tool to successfully read and follow the documentation, compile the tool and obtain the expected result when executing an analysis of a MWE. We sent an email to the authors of each tool to confirm that we used the more suitable version of the code, that the command line we used to analyze an application is the most suitable one and, in some cases, requested some help to solve issues in the building process. We reported in Table 2 the authors that answered our request and confirmed our decisions.

From this building phase, several observations can be made. Using a recent operating system, it is almost impossible in a reasonable amount of time to rebuild a tool released years ago. Too many dependencies, even for Java based programs, trigger compilation or execution problems. Thus, if the documentation mentions a specific operating system, we use a Docker image of this OS. Most of the time, tools require additional external components to be fully functional. It could be resources such as the android.jar file for each version of the SDK, a database, additional libraries or tools. Depending of the quality of the documentation, setting up those components can take hours to days. This is why we automatized in a Dockerfile the setup of the environment in which the tool is built and run[3].

[3] To guarantee reproducibility we published the results, datasets, Dockerfiles and containers. Source code is located at: https://github.com/histausse/rasta, datasets and experiments results are available at: https://zenodo.org/records/10144014, Singularity containers are available at: https://zenodo.org/records/10980349, Docker images (`histausse/rasta-<toolname>:icsr2024`) can be downloaded from Docker Hub.

3.3 Runtime Conditions

As shown in Fig. 1, before benchmarking the tools, we built and installed them in a Docker containers for facilitating any reuse of other researchers. We converted them into Singularity containers because we had access to such a cluster and because this technology is often used by the HPC community for ensuring the reproducibility of experiments. We performed manual tests using these Singularity images to check:

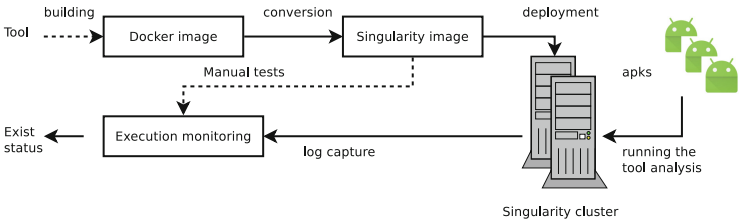

Fig. 1. Methodology overview

- the location where the tool is writing on the disk. For the best performances, we expect the tools to write on a mount point backed by an SSD. Some tools may write data at unexpected locations which required small patches from us.
- the amount of memory allocated to the tool. We checked that the tool could run a MWE with a 64 GB limit of RAM.
- the network connection opened by the tool, if any. We expect the tool not to perform any network operation such as the download of Android SDKs. Thus, we prepared the required files and cached them in the images during the building phase. In a few cases, we patched the tool to disable the download of resources.

A campaign of tests consists in executing the 20 selected tools on all APKs of a dataset. The constraints applied on the clusters are:

- No network connection is authorized in order to limit any execution of malicious software.
- The allocated RAM for a task is 64 GB.
- The allocated maximum time is 1 h.
- The allocated object space / stack space is 64 GB / 16 GB if the tool is a Java based program.

For the disk files, we use a mount point that is stored on a SSD disk, with no particular limit of size. Note that, because the allocation of 64 GB could be insufficient for some tool, we evaluated the results of the tools on 20% of our dataset (described later in Sect. 3.4) with 128 GB of RAM and 64 GB of RAM and checked that the results were similar. With this confirmation, we continued our evaluations with 64 GB of RAM only.

3.4 Dataset

We built a dataset named **Rasta** to cover all dates between 2010 to 2023. This dataset is a random extract of Androzoo [1], for which we balanced applications between years and size. For each year and inter-decile range of size in Androzoo, 500 applications have been extracted with an arbitrary proportion of 7% of malware. This ratio has been chosen because it is the ratio of goodware/malware that we observed when performing a raw extract of Androzoo. For checking the maliciousness of an Android application we rely on the VirusTotal detection indicators. If more than 5 antiviruses have flagged the application as malicious, we consider it as a malware. If no antivirus has reported the application as malicious, we consider it as a goodware. Applications in between are dropped.

For computing the release date of an application, we contacted the authors of Androzoo to compute the minimum date between the submission to Androzoo and the first upload to VirusTotal. Such a computation is more reliable than using the DEX date that is often obfuscated when packaging the application.

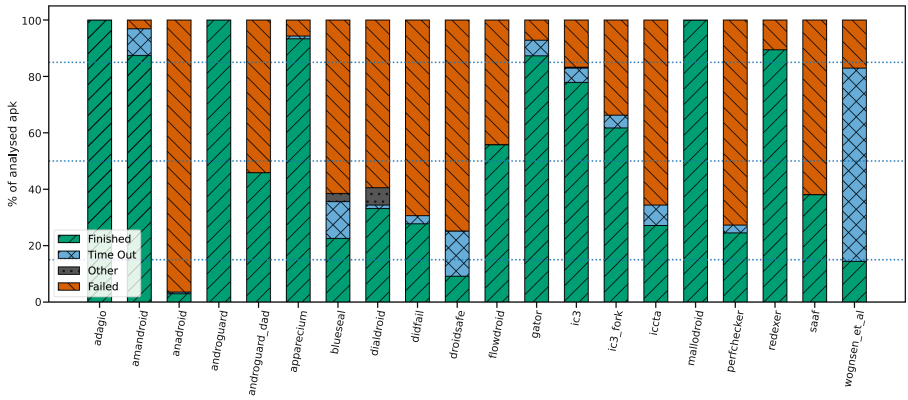

Fig. 2. Exit status for the Rasta dataset

4 Experiments

4.1 RQ1: Re-usability Evaluation

Figure 2 represents the success/failure rate (green/orange) of the tools. We distinguished failure to compute a result from timeout (blue) and crashes of our evaluation framework (in grey, probably due to out of memory kills of the container itself). Because it may be caused by a bug in our own analysis stack, exit status represented in grey (Other) are considered as unknown errors and not as failure of the tool.

We observe a global increase of the failed status: 12 tools (54.5%) have a finishing rate below 50%. Three tools (androguard_dad, blueseal, saaf) reach the bar of 50% of failure. 7 tools keep a high success rate: Adagio, Amandroid, Androguard, Apparecium, Gator, Mallodroid, Redexer. Regarding IC3, the fork with a simpler build process and support for modern OS has a lower success rate than the original tool.

Two tools should be discussed in particular. Androguard has a high success rate which is not surprising: it used by a lot of tools, including for analyzing application uploaded to the Androzoo repository. Nevertheless, when using Androguard decompiler (DAD) to decompile an APK, it fails more than 50% of the time. This example shows that even a tool that is frequently used can still run into critical failures. Concerning Flowdroid, our results show a very low timeout rate (0.06%) which was unexpected: in our exchanges, Flowdroid's author were expecting a higher rate of timeout and fewer crashes.

As a summary, the final ratio of successful analysis for the tools that we could run is 54.9%. When including the two defective tools, this ratio drops to 49.9%.

RQ1 answer: On a recent dataset we consider that 54.5% of the tools are unusable. For the tools that we could run, 54.9% of analysis are finishing successfully.

4.2 RQ2: Size, SDK and Date Influence

To measure the influence of the date, SDK version and size of applications, we fixed one parameter while varying an other. For the sake of clarity, we separated Java based / non Java based tools.

Fixed application year. (5000 APKs) We selected the year 2022 which has a good amount of representatives for each decile of size in our application dataset. Figures 3 (resp. 4) shows the finishing rate of the tools in function of the size of the bytecode for Java based tools (resp. non Java based tools) analyzing applications of 2022. We can observe that all Java based tools have a finishing rate decreasing over years. 50% of non Java based tools have the same behavior.

Fixed application bytecode size. (6252 APKs) We selected the sixth decile (between 4.08 and 5.20 MB), which is well represented in a wide number of years. Figures 5 and 6 represent the finishing rate depending of the year at a fixed bytecode size. We observe that 9 tools over 12 have a finishing rate dropping below 20% for Java based tools, which is not the case for non Java based tools.

We performed similar experiments by variating the min SDK and target SDK versions, still with a fixed bytecode size between 4.08 and 5.2 MB, as shown in Fig. 7 and 8. We found that contrary to the target SDK, the min SDK version has an impact on the finishing rate of Java based tools: 8 tools over 12 are below 50% after SDK 16. It is not surprising, as the min SDK is highly correlated to the year.

RQ2 answer: The success rate varies based on the size of bytecode and SDK version. The date is correlated with the success rate for Java based tools only.

4.3 RQ3: Malware Vs Goodware

We compared the finishing rate of malware and goodware applications for evaluated tools. Because, the size of applications impacts this finishing rate, it is interesting to compare the success rate for each decile of bytecode size. Table 3 reports the bytecode size and the finishing rate of goodware and malware in each decile of size. We also computed the ratio of the bytecode size and finishing rate for the two populations. We observe that the ratio for the finishing rate decreases from 1.04 to 0.73, while the ratio of the bytecode size is around 1. We conclude from this table that analyzing malware triggers less errors than for goodware.

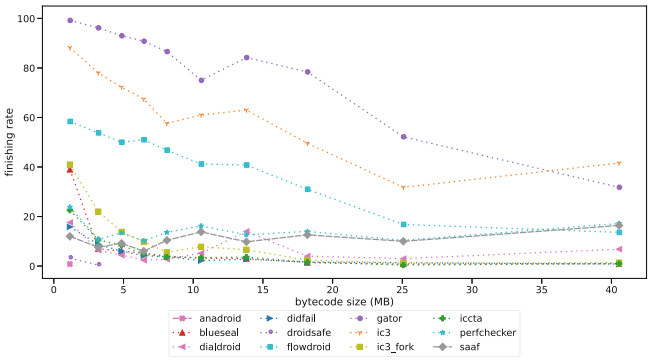

Fig. 3. Finishing rate by bytecode size for APK detected in 2022 – **Java based tools**

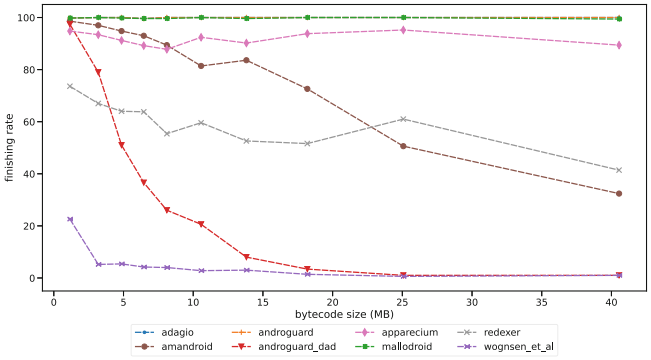

Fig. 4. Finishing rate by bytecode size for APK detected in 2022 – **Non Java based tools**

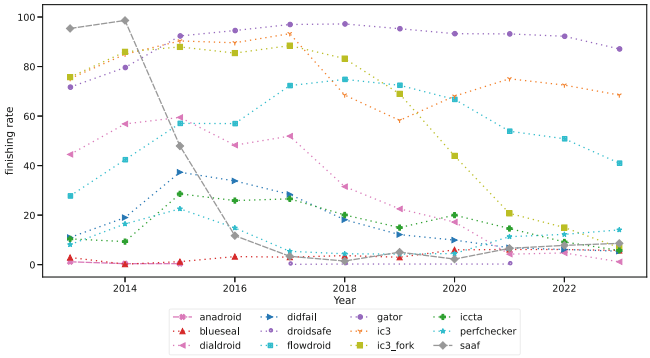

Fig. 5. Finishing rate by discovery year with a bytecode size ∈ [4.08, 5.2] MB – **Java based tools**

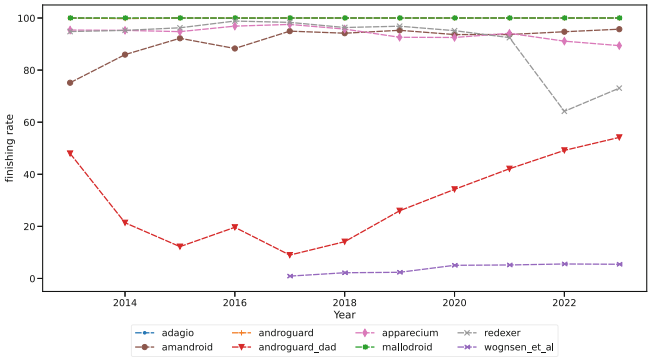

Fig. 6. Finishing rate by discovery year with a bytecode size ∈ [4.08, 5.2] MB – **Non Java based tools**

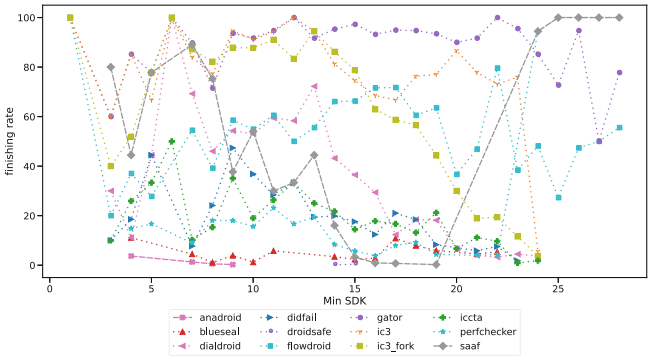

Fig. 7. Finishing rate by min SDK with a bytecode size ∈ [4.08, 5.2] MB – **Java based tools**

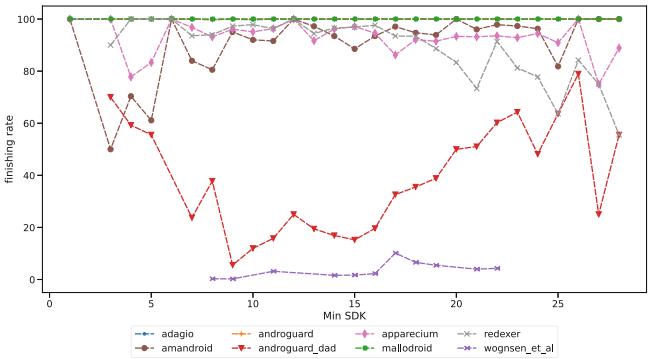

Fig. 8. Finishing rate by min SDK with a bytecode size ∈ [4.08, 5.2] MB – **Non Java based tools**

RQ3 answer: Analyzing malware applications triggers less errors for static analysis tools than analyzing goodware for comparable bytecode size.

Table 3. DEX size and Finishing Rate (FR) per decile

Decile	Avg DEX size MB		Finishing Rate: FR		Ratio Size	Ratio FR
	Good	Mal	Good	Mal	Good/Mal	Good/Mal
1	0.13	0.11	0.85	0.82	1.17	1.04
2	0.54	0.55	0.74	0.72	0.97	1.03
3	1.37	1.25	0.63	0.66	1.09	0.97
4	2.41	2.34	0.57	0.62	1.03	0.92
5	3.56	3.55	0.53	0.59	1.00	0.90
6	4.61	4.56	0.50	0.61	1.01	0.82
7	5.87	5.91	0.47	0.57	0.99	0.83
8	7.64	7.63	0.43	0.56	1.00	0.76
9	11.39	11.26	0.39	0.58	1.01	0.67
10	24.24	21.36	0.33	0.46	1.13	0.73

5 Discussion

5.1 State-of-the-Art Comparison

Our finding are consistent with the numerical results of Pauck *et al.* that showed that 58.9% of DIALDroid-Bench [5] real-world applications are analyzed successfully with the 6 evaluated tools [25]. Six years after the release of DIALDroid-Bench, we obtain a lower ratio of 40.1% for the same set of 6 tools but using

the Rasta dataset of 62 525 applications. We extended this result to a set of 20 tools and obtained a global success rate of 54.9%. We confirmed that most tools require a significant amount of work to get them running [28].

Investigating the reason behind tools' errors is a difficult task and will be investigated in a future work. For now, our manual investigations show that the nature of errors varies from one analysis to another, without any easy solution for the end user for fixing it.

5.2 Recommendations

Finally, we summarize some takeaways that developers should follow to improve the success of reusing their developed software.

For improving the reliability of their software, developers should use classical development best practices, for example continuous integration, testing, code review. For improving the reusability developers should write a documentation about the tool usage and provide a minimal working example and describe the expected results. Interactions with the running environment should be minimized, for example by using a docker container, a virtual environment or even a virtual machine. Additionally, a small dataset should be provided for a more extensive test campaign and the publishing of the expected result on this dataset would ensure to be able to evaluate the reproducibility of experiments.

Finally, an important remark concerns the libraries used by a tool. We have seen two types of libraries: a) internal libraries manipulating internal data of the tool; b) external libraries that are used to manipulate the input data (APKs, bytecode, resources). We observed by our manual investigations that external libraries are the ones leading to crashes because of variations in recent APKs (file format, unknown bytecode instructions, multi-DEX files). We believe that the developer should provide enough documentation to make possible a later upgrade of these external libraries.

5.3 Threats to Validity

Our application dataset is biased in favor of Androguard, because Androzoo have already used Androguard internally when collecting applications and discarded any application that cannot be processed with this tool.

Despite our best efforts, it is possible that we made mistakes when building or using the tools. It is also possible that we wrongly classified a result as a failure. To mitigate this possible problem we contacted the authors of the tools to confirm that we used the right parameters and chose a valid failure criterion.

The timeout value, amount of memory are arbitrarily fixed. For mitigating their effect, a small extract of our dataset has been analyzed with more memory/time for measuring any difference.

Finally, the use of VirusTotal for determining if an application is a malware or not may be wrong. For limiting this impact, we used a threshold of at most 5 antiviruses (resp. no more than 0) reporting an application as being a malware (resp. goodware) for taking a decision about maliciousness (resp. benignness).

6 Conclusion

This paper has assessed the suggested results of the literature [22,25,28] about the reliability of static analysis tools for Android applications. With a dataset of 62 525 applications we established that 54.5% of 22 tools are not reusable, when considering that a tool that has more than 50% of time a failure is unusable. In total, the analysis success rate of the tools that we could run for the entire dataset is 54.9%. The characteristics that have the most influence on the success rate is the bytecode size and min SDK version. Finally, we showed that malware APKs have a better finishing rate than goodware.

In future works, we plan to investigate deeper the reported errors of the tools in order to analyze the most common types of errors, in particular for Java based tools. We also plan to extend this work with a selection of more recent tools performing static analysis.

Acknowledgments. This work was supported by the ANR Research under the Plan France 2030 bearing the reference ANR-22-PECY-0007.

References

1. Allix, K., Bissyandé, T.F., Klein, J., Traon, Y.L.: AndroZoo: collecting millions of android apps for the research community. In: 13th Working Conference on Mining Software Repositories (MSR), pp. 468–471 (2016)
2. Arzt, S., et al.: FlowDroid: precise context, flow, field, objectsensitive and lifecycle-aware taint analysis for android apps. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 259–269. ACM Press, Edinburgh, UK (2014). https://doi.org/10.1145/2666356.2594299
3. Azim, T., Neamtiu, I.: Targeted and depth-first exploration for systematic testing of android apps. In: Hosking, A.L., Eugster, P.T., and Lopes, C.V. (eds.) Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, Part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013, pp. 641–660. ACM (2013). https://doi.org/10.1145/2509136.2509549
4. Blackshear, S., Chang, B.-Y.E., Sridharan, M.: Thresher: precise refutations for heap reachability. SIGPLAN Not. **48**(6), 275–286 (2013). https://doi.org/10.1145/2499370.2462186
5. Bosu, A., Liu, F., Yao, D., Wang, G.: Collusive data leak and more: large-scale threat analysis of inter-app communications. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, pp. 71–85. ACM, Abu Dhabi United Arab Emirates (2017). https://doi.org/10.1145/3052973.3053004
6. Choi, K., and Chang, B.-M.: A type and effect system for activation flow of components in android programs. Inform. Process. Lett. **114**(11), 620–627 (2014). https://doi.org/10.1016/j.ipl.2014.05.011
7. Desnos, A., Gueguen, G.: Android: From Reversing to Decompilation. Black Hat Abu Dhabi (2011)

8. Enck, W., et al.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: 9th USENIX Symposium on Operating Systems Design and Implementation, pp. 393–407. USENIX Association, Vancouver, BC, Canada (2010)
9. Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., Smith, M.: Why Eve and Mallory love android: an analysis of android SSL (in) security. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 50–61. ACM, Raleigh North Carolina USA (2012). https://doi.org/10.1145/2382196.2382205
10. Gascon, H., Yamaguchi, F., Arp, D., Rieck, K.: Structural detection of android malware using embedded call graphs. In: Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security, pp. 45–54. ACM, Berlin Germany (2013). https://doi.org/10.1145/2517312.2517315
11. Geneiatakis, D., Fovino, I.N., Kounelis, I., Stirparo, P.: A permission verification approach for android mobile applications. Comput. Security **49**, 192–205 (2015). https://doi.org/10.1016/j.cose.2014.10.005
12. Gordon, M.I., Kim, D., Perkins, J.H., Gilham, L., Nguyen, N., Rinard, M.C.: Information flow analysis of android applications in droidsafe. In: 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8–11, 2015. The Internet Society (2015)
13. Hoffmann, J., Ussath, M., Holz, T., Spreitzenbarth, M.: Slicing droids: program slicing for smali code. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. SAC '13, pp. 1844–1851. Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2480362.2480706
14. Jeon, J., et al.: Dr. Android and Mr. Hide: fine-grained permissions in android applications. In: Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 3–14. ACM, Raleigh North Carolina USA (2012). https://doi.org/10.1145/2381934.2381938
15. Klieber, W., Flynn, L., Bhosale, A., Jia, L., Bauer, L.: Android taint flow analysis for app sets. In: Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis, pp. 1–6. ACM, Edinburgh United Kingdom (2014). https://doi.org/10.1145/2614628.2614633
16. Li, L., et al.: IccTA: detecting inter-component privacy leaks in android apps. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, pp. 280–291. IEEE, Florence, Italy (2015). https://doi.org/10.1109/ICSE.2015.48
17. Federrath, H., Gollmann, D. (eds.): ICT Systems Security and Privacy Protection: 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings. Springer International Publishing, Cham (2015)
18. Li, L.: Static analysis of android apps: a systematic literature review. Inform. Softw. Technol. **88**, 67–95 (2017). https://doi.org/10.1016/j.infsof.2017.04.001
19. Liang, S., et al.: Sound and precise malware analysis for android via pushdown reachability and entry-point saturation. In: Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones and Mobile Devices. SPSM '13, pp. 21–32. Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2516760.2516769
20. Lillack, M., Kästner, C., Bodden, E.: Tracking load-time configuration options. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. ASE '14, pp. 445–456. Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2642937.2643001

21. Liu, Y., Xu, C., Cheung, S.-C.: Characterizing and detecting performance bugs for smartphone applications. In: Proceedings of the 36th International Conference on Software Engineering, pp. 1013–1024. ACM, Hyderabad India (2014). https://doi.org/10.1145/2568225.2568229
22. Luo, L., et al.: TaintBench: automatic real-world malware benchmarking of android taint analyses. Empir. Softw. Eng. **27**(1), 16 (2022). https://doi.org/10.1007/s10664-021-10013-5
23. Octeau, D., Luchaup, D., Dering, M., Jha, S., McDaniel, P.: Composite constant propagation: application to android inter-component communication analysis. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, pp. 77–88. IEEE, Florence, Italy (2015). https://doi.org/10.1109/ICSE.2015.30
24. Octeau, D., et al.: Effective Inter-Component communication mapping in android: An essential step towards holistic security analysis. In: 22nd USENIX Security Symposium (USENIX Security 13), pp. 543–558 (2013)
25. Pauck, F., Bodden, E., Wehrheim, H.: Do android taint analysis tools keep their promises? In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 331–341. ACM, Lake Buena Vista FL USA (2018). https://doi.org/10.1145/3236024.3236029
26. Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., Cavallaro, L.: TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time (2018)
27. Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., Vigna, G.: Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23–26, 2014. The Internet Society (2014)
28. Reaves, B., et al.: *droid: assessment and evaluation of android application analysis tools. ACM Comput. Surv. **49**(3), 55:1–55:30 (2016). https://doi.org/10.1145/2996358
29. Rountev, A., Yan, D.: Static reference analysis for GUI objects in android software. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, pp. 143–153. ACM, Orlando FL USA (2014). https://doi.org/10.1145/2544137.2544159
30. Shen, F., et al.: Information flows as a permission mechanism. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, pp. 515–526. ACM, Vasteras Sweden (2014). https://doi.org/10.1145/2642937.2643018
31. Titze, D., Schutte, J.: Apparecium: revealing data flows in android applications. In: 2015 IEEE 29th International Conference on Advanced Information Networking and Applications, pp. 579–586. IEEE, Gwangiu, South Korea (2015). https://doi.org/10.1109/AINA.2015.239
32. Vidas, T., Tan, J., Nahata, J., Tan, C.L., Christin, N., Tague, P.: A5: Automated analysis of adversarial android applications. In: Proceedings of the 4th ACMWorkshop on Security and Privacy in Smartphones & Mobile Devices, pp. 39–50. ACM, Scottsdale Arizona USA (2014). https://doi.org/10.1145/2666620.2666630
33. Wei, F., Roy, S., Ou, X., Robby: amandroid: a precise and general intercomponent data flow analysis framework for security vetting of android apps. In: ACM SIGSAC Conference on Computer and Communications Security, pp. 1329–1341. ACM, Scottsdale Arizona USA (2014). https://doi.org/10.1145/2660267.2660357

34. Wognsen, E.R., Karlsen, H.S., Olesen, M.C., Hansen, R.R.: Formalisation and analysis of dalvik bytecode. Sci. Comput. Programm. **92**, 25–55 (2014). https://doi.org/10.1016/j.scico.2013.11.037
35. Xia, M., Gong, L., Lyu, Y., Qi, Z., Liu, X.: Effective real-time android application auditing. In: 2015 IEEE Symposium on Security and Privacy, pp. 899–914. IEEE, San Jose, CA (2015). https://doi.org/10.1109/SP.2015.60
36. Yang, S., Yan, D., Wu, H., Wang, Y., Rountev, A.: static control-flow analysis of user-driven callbacks in android applications. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, pp. 89–99. IEEE, Florence, Italy (2015). https://doi.org/10.1109/ICSE.2015.31
37. Zhauniarovich, Y., Ahmad, M., Gadyatskaya, O., Crispo, B., Massacci, F.: StaDynA: addressing the problem of dynamic code updates in the security analysis of android applications. In: Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, pp. 37–48. ACM, San Antonio Texas USA (2015). https://doi.org/10.1145/2699026.2699105

The Current Status of Open Source ERP Systems: A GitHub Analysis

Georgia M. Kapitsaki(✉) and Maria Papoutsoglou

Department of Computer Science, University of Cyprus, Nicosia, Cyprus
gkapi@ucy.ac.cy

Abstract. Enterprise resource planning (ERP) systems are widely used by larger and smaller enterprises to assist them in managing and utilizing data coming from different business operations. Although many ERPs are based on commercial solutions, there are many open source alternatives that are also being widely adopted by organizations (e.g., *Odoo*). Their characteristics and evolution as captured in online version control systems have not been investigated but they are useful as an initial step in order to understand reuse practices. In this work, we are examining the current state in the development of such open source ERP systems. We used GitHub as source of data of open source ERP systems and analyzed 11,500 open source ERP repositories that we collected. We investigated the volume of ERP relevant repositories over the years, their popularity, reuse in other ERP repositories and point out some characteristics of popular ERP systems. We observed that there is a sharp increase in the number of created repositories in recent years, while the large majority of them does not indicate an open source software license.

Keywords: Enterprise resource planning · GitHub · software reuse · empirical study

1 Introduction

Enterprise resource planning (ERP) systems allow organizations to manage a number of their main business functions, including human resources, supply chain management, sales and customer relationship management (CRM). According to Acumen Research and Consulting, the global market for ERP software has seen significant and is expected to grow even more within the next decade (from 53.2 USD billion in 2022 to 127.7 USD billion in 2032[1]). ERP systems used to focus more on the activities of large enterprises but nowadays they are also used by smaller organizations. Although many ERP systems are commercial and available with a proprietary license, there are many open source software (OSS) alternatives, an overview of which is provided by Mladenova [11]. The advantages of using open source ERP systems are indicated including the

[1] https://www.acumenresearchandconsulting.com/erp-software-market.

A. Achilleos et al. (Eds.): ICSR 2024, LNCS 14614, pp. 171–188, 2024.
https://doi.org/10.1007/978-3-031-66459-5_11

cost advantage and the possibility to modify the source code following the organization's needs. Especially for smaller enterprises it may be more meaningful to invest in such solutions in order to save resources [15].

Previous works have focused on the evolution of ERP systems over the years focusing on their type and market position [17] or have examined how specific ERP systems are updated over the years, e.g., for the case of cloud-based enterprise resource planning [1]. These works are relying on the study of the existing literature or of a specific system and its context but it is also interesting to study the available open source ERP systems using data from existing open source systems as an initial step towards understanding reuse practices. By mining GitHub repositories, researchers or developers can capture evolving trends in ERP software development, such as frequent updates, the introduction of new features, or shifts in technology stacks or the adoption of different licenses during years. GitHub is the most suitable source to collect such data, as it is widely used by many organizations and hosts many popular OSS projects [5].

In this work we focus on examining the presence of ERPs in GitHub repositories. Generally examining the landscape of available GitHub repositories in the ERPs could reveal an increasing trend in repository creation, especially among OSS ERP systems on GitHub and can serve as a rich data source for understanding the dynamics of software reuse. It can serve as a stepping stone towards an empirical framework for identifying common components and modules that are frequently reused across different projects. This can lead to more efficient development practices, as common functionalities do not need to be recreated from scratch. Instead, developers can focus on adapting and integrating these pre-existing, well-tested components into their projects, which can significantly reduce development time and costs. But in order to find these pathways in the empirical research we need to gain first insights for the landscape of ERP GitHub projects.

Using the GitHub Search API for data collection and after performing appropriate filtering, we have used data from 11,500 relevant software repositories and have analyzed their metadata, using the repository information returned by GitHub. Our results show that there is a sharp increase in the number of created repositories in recent years, the large majority of them does not indicate an open source software license, and there are some popular repositories that dominate and have built independent communities around them. We examine the following Research Questions (RQs):

- **RQ1.** *How have open source ERP systems evolved over the years?* In this RQ, we are examining how many ERP systems have emerged over the years, and whether they are still maintained. This is an initial RQ to gain an understanding of ERPs available on GitHub.
- **RQ2.** *How popular are open source ERP systems and which are the most popular ones?* We used mainly the repository forks count and additionally star/watchers count to determine the impact of each repository. As forks are a way for users to modify their own copy of a repository, they can be used as

a proxy for counting reuse. We also examined whether the type of repository owner affects the system's popularity.
- **RQ3.** *Which are the most popular programming languages and the usual licenses in open source ERP systems?* The main programming language used in each software repository was analyzed in this RQ, along with the open source software license assigned to the repository (e.g., MIT, GPL-2.0, etc.). The licensing scheme of an OSS is important for its further use and distribution, and may affect subsequent reuse [6].
- **RQ4.** *Which repositories build upon other repositories?* Many repositories are add-ons to existing ERPs, so they are reusing the basic repository to provide additional functionality. In this RQ, we looked into the repository description to examine whether additional popular repositories from the dataset are indicated.

We argue that the current work is useful for observing the current trends in open source ERP systems, as reflected in GitHub repositories and understand basic reuse practices (i.e., using most popular repositories in terms of reuse counting forks and appearances of existing repositories in other repositories). It can also assist organizations active in OSS ERP systems to better place their systems to foster reuse, as it shows some basic characteristics of popular repositories.

The rest of the paper is structured as follows. Section 2 presents relevant works in the area, whereas Sect. 3 is dedicated to the methodological process followed. The results for each RQ are presented and discussed in Sect. 4. Section 5 discusses main take away messages, while Sect. 6 is dedicated to the study limitations. Finally, Sect. 7 concludes the paper outlining also directions of future work.

2 Related Work

The feasibility of OSS ERP systems for small businesses was examined by Olson and Staley [15]. The authors describe specifically the adoption case in a small enterprise that manufactures industrial automated welding systems. The decision process is described leading to the use of *ERPLite* and at a later stage, as the business needs changed, to *xTuple*. The two systems were compared using six dimensions (organizational, business-related, technological, entrepreneurial, contractual and financial). It was concluded that time for configurations is required but on the positive side organizations have the opportunity to tune the software around business practices that provide a competitive advantage. As new domains emerge in ERP systems, special requirements may apply. For instance, when modeling socio-cyber-physical systems not all well-known enterprise architecture modeling languages may be suitable [9]. This previous work analyzed and discussed the challenges of ArchiMate, Industry 4.0 Reference Architecture Model RAMI 4.0 and St. Alter's Work Systems framework.

Open platforms as an advantage for businesses is investigated by Jazayeri et al. with the aim to capture variabilities in the design of open platforms and their

relation to business objectives [7]. A pattern-driven approach, called SecoArc, is suggested for this purpose based on the characteristics of 111 open platforms. When it comes to ERP systems, an agile approach for ERP selection that uses the strengths of service oriented ERP can be found in AMES (Agile Method for ERP Selection) [8]. The method that includes a number of iterative steps was adopted at a small entrepreneurial firm.

GitHub is used as a data source in a variety of works, e.g., in order to investigate the developers' expertise [4], trends in green economy such as the Electric Vehicles sector [18] or to investigate what makes a repository popular [2]. Forking practices in social coding have also been examined [23]. It was found that inefficiencies exist, such as lost contributions, redundant development and fragmented communities, whereas modularity and centralized management can assist in this process. Domain analysis on GitHub has also been addressed in previous works. The automotive software landscape using data from GitHub was analyzed and it was found that more than 15,000 users contribute to approximately 600 actively-developed automotive software projects between 2010 and 2021 [10]. It was also found that one out of three automotive software repositories is owned by an organization and that overall automotive repositories are less popular than general purpose repositories. In the area of bioinformatics, 1,720 repositories from GitHub that were mentioned in bioinformatics articles were used to investigate the relationships between code properties, development activity, developer communities, and software impact [19]. Various analysis techniques were used, including topic modeling and gender analysis.

Software reuse involves utilizing existing components, known as reusable assets, to develop new systems, offering benefits like reduced development efforts and improved quality. However, few methods exist for extracting these assets. A method called JReuse identifies reuse opportunities by analyzing naming similarities of object-oriented entities like classes and methods [13,14]. The authors have developed a repository by evaluating similarly named entities across various systems and validate their approach with 38 e-commerce information systems from GitHub, demonstrating JReuse's effectiveness in identifying relevant classes and methods.

Relation to Previous Works. Some previous works have analyzed open source ERP systems but they have focused on a limited number of systems and have relied on manual analysis in order to draw conclusions [11]. In this work, we focus on a large scale analysis of existing systems using data from public repositories on GitHub. To the best of our knowledge, no previous work has investigated the open source ERP community as reflected on GitHub with a focus on reuse.

3 Methodology

The steps followed in the methodology of this work are depicted in Fig. 1. We have gathered relevant repositories from GitHub by using the GitHub Search API. Specifically, we have gathered all repositories that include the terms *erp* or *enterprise resource planning*. This search allows us to also retrieve repositories

that do not have the above terms in their name, but indicate the term nevertheless (for example in the repository topics). GitHub Search API returns 41,467 repositories containing the term *erp* and 592 containing the term *enterprise resource planning*. In order to ensure that the dataset includes active repositories, we have applied the following exclusion/inclusion criteria, as also performed similarly in previous works [10]:

- **Size.** Repositories have to have a size or more than 0kB.
- **Data availability.** The repository information needed to be available and be returned by the GitHub API.
- **Forks.** We excluded repositories that are forks, to avoid adding the same repository twice in the dataset.

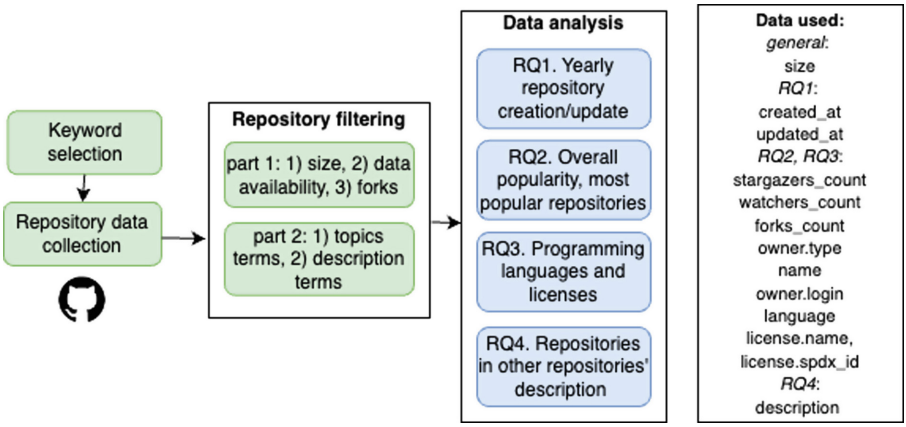

Fig. 1. Steps of the methodological process.

After the above, 33,771 repositories remained. In order to verify that the above repositories are valid cases of ERP systems, we manually went through the README.md files and the topics of 100 randomly selected repositories that did not include the search terms in the repository name. This number was considered sufficient, as 6,126 repositories do not contain the terms *erp*, *enterprise* or *resource* in their name. We verified that they refer indeed to an open source ERP system, apart from 5 repositories (*Some-Many-Books*, *Books-Free-Books*, *EEG-tools-and-tips*, *UnfoldSim.jl*, *texpovn*) that we removed from the dataset. We also went through the 100 repositories with most stars/watchers and found 3 irrelevant repositories (*erpc*, *Aries*, *likeshop*) that we also removed. Since irrelevant repositories were found via this manual verification, we performed additional filtering on the repositories relying on the observations made during the manual inspection of the 200 repositories. We filtered the repositories keeping the repositories that satisfy at least one of the following: 1) are using the term *erp* in their topics (there are 2,091 repositories in the dataset that are using topics),

or 2) are indicating one of the following terms in their description: *enterprise*, *resource*, *planning*, *erp* with at least one space before or after the term.

The size of the dataset after applying all the above filtering steps is shown in Table 1 (in the *After filtering* column). The table shows also the approximate values of the repository size in MB (GitHub provides the size of the repository in kB). Most repositories are smaller than 1 MB. The dataset is available for replication purposes online on Zenodo.[2]

Table 1. Dataset size.

Keyword	Collected	After filtering (no duplicates)	Max size	Avg. size	Median size
erp	41,467				
enterprise resource planning	592				
TOTAL	42,059	**11,500**	7,441 MB	19.47 MB	0.69 MB

4 Results and Discussion

4.1 RQ1. Evolution over the Years

We used the repository's created date (*created_at*) and update date (*updated_at*) to extract the relevant years and counted how many repositories have been created per year and are still maintained. Figure 2 depicts the repositories created per year since 2008 that is the year GitHub was launched. Although there were only 2 new repositories in 2008, the number of repositories created is larger every year with a drop only in 2021. Excluding 2021 the increase is especially visible in 2017 and onward, reaching a very high number in 2023 with 1,924 new repositories. Some repositories refer to different parts or add-ons of the same system (as we observed for repositories coming from the Odoo Community Association - OCA organization), so the repository numbers may not correspond to completely different ERP systems, but this does not change the growth rate tendency. The creation of new repositories linked with add-ons of ERP systems shows a trend towards reusing specific existing ERPs. Overall, the increase in the number of created repositories is aligned with the market growth of ERP systems as reported by Acumen Research and Consulting.

Using *updated_at* field we see that 3,709 repositories (approximately one third or 32.25% of all repositories), are still maintained, i.e., have a commit performed in the year 2023. If we consider only the 9,577 repositories created earlier than 2023, this number drops to 1,787 or 15.54% of all repositories. So although a very large number of repositories have been created, only some of them manage to last over the years. This may mean that they managed to form a community that contributes to their maintenance or have seen acceptance to a level that makes their maintenance by the original repository creator feasible. It is however, not an indicator of reuse but repository use.

[2] https://zenodo.org/records/11198589.

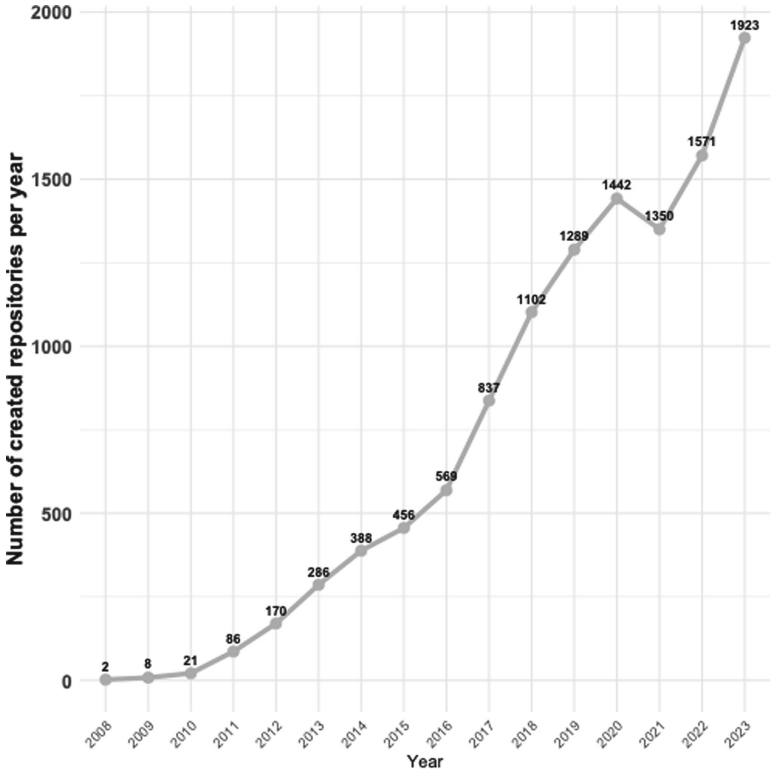

Fig. 2. Created repositories per year.

4.2 RQ2. Popularity of Open Source ERP Systems

Our dataset includes popular open source ERPs, such as *ADempiere*[3] (12th most forked in dataset) and *Dolibarr*[4] (3rd most forked in dataset), including all but one of the 16 systems listed in [11]. We examined nevertheless the characteristics of popularity and reuse indicators within the dataset, using as proxies primarily the number of forks and partially stars/watchers count. The number of stars has been used as a proxy for repository popularity in works in the past [2]. Forks have also been used to count repository popularity but, apart from a way to contribute back to the repository, they are used as a proxy for software reuse, as users may fork a repository in order to create their own copy that they modify [3,20]. Forking is also a way to establish fork ecosystems that, although similar, are developed in parallel [12].

There are 7.81 forks on average for each repository, whereas stars/watchers (the value of stars and watchers is the same for both indications in all repositories) is 11.14. The number of stars is relatively low, considering that trending

[3] http://www.adempiere.net/web/guest/welcome.
[4] https://www.dolibarr.org/.

repositories in GitHub have 30 stars or more.[5] The top 40 repositories in the dataset in terms of forks count are listed in Table 2, along with the main programming language of the repository and its OSS license. For the software license names the notation of the Software package data exchange (SPDX) specification is used. We are also indicating in the table the number of contributors as retrieved directly from the repositories pages on GitHub (this information was not part of the original dataset). The first two repositories (*Odoo*[6] and *ERPNext*[7]) have a much larger number of stars and forks than the third one), indicating that these are the dominant open source ERP systems. *Odoo* is an open source suite that includes different applications (e.g., CRM, warehouse management). In the dataset, we find also repositories referring to add-ons or extensions to existing ERPs, whereas some repositories are in languages other than English that we kept however in the dataset as valid ERP system cases. For instance, the 6th top repository is the repository of the official Docker image for Odoo, whereas a large number of popular repositories comes from the Odoo Community Association organization.

We investigated whether the type of the repository (owned by a user or an organization) affects its popularity. Overall, the vast majority of repositories in the dataset are user owned: the dataset contains 1,601 organization owned repositories and 9,899 user owned repositories. Using an independent samples t-test to test the hypothesis that organization owned repositories are more popular (with the forks count as dependent and the repository owner type as independent variable), a statistically significant different was found, verifying our hypothesis ($p-value < 0.01$, $F = 206.045$), with organization owned repositories having on average 47.63 forks compared to 1.37 forks for user owned repositories. So generally, organization owned repositories are trusted more by users, they contribute more to them and copy them in their own forks for modification purposes. The same hypothesis stands when stars/watchers are considered: organization owned repositories have on average 66.65 stars compared to 2.17 stars for user owned repositories ($p - value < 0.01$, $F = 150.001$).

4.3 RQ3. Programming Languages and Licensing Schemes

In each GitHub repository, the percentages of programming languages used in the repository's source code are provided, with one programming language indicated as main in the *language* field returned by the GItHub Search API. 115 different programming languages as main language of the repository are found in the dataset. Limiting the presentation to the main programming languages in the dataset, Table 3 shows the top 20 programming languages that appear as main in repositories. Python, Java, JavaScript, C# and PHP were also among the programming languages encountered in the 16 OSS ERP systems analyzed in [11]. This previous work also lists the following languages: C++, Scala, Kotlin,

[5] https://github.com/trending.
[6] https://www.odoo.com/.
[7] https://erpnext.com/.

Table 2. 40 open source ERP repositories mostly forked.

	Repository	Owner	% forks	# stars/ watchers	Contributors	Language	License
1	*odoo*	odoo	20,775	31,845	1,879	Python	LGPL-3.0
2	*erpnext*	frappe	6,099	15,360	530	Python	GPL-3.0
3	*dolibarr*	Dolibarr	2,483	4,481	598	PHP	GPL-3.0
4	*akaunting*	akaunting	2,199	6,929	92	PHP	BUSL-1.1
5	*web*	OCA	1,826	828	224	JavaScript	AGPL-3.0
6	*docker*	odoo	1,470	839	17	Dockerfile	LGPL-3.0
7	*server-tools*	OCA	1,383	600	196	Python	AGPL-3.0
8	*idurar-erp-crm*	idurar	921	3,978	73	JavaScript	various licenses
9	*sale-workflow*	OCA	921	258	279	HTML	AGPL-3.0
10	*scm-biz-suite*	doublechaintech	891	2,293	3	Java	Apache-2.0
11	*partner-contact*	OCA	778	177	233	HTML	AGPL-3.0
12	*adempiere*	adempiere	761	750	30	Java	GPL-2.0
13	*reporting-engine*	OCA	728	280	151	Python	AGPL-3.0
14	*project*	OCA	721	237	129	HTML	AGPL-3.0
15	*account-financial-tools*	OCA	711	264	179	Python	AGPL-3.0
16	*purchase-workflow*	OCA	702	172	223	HTML	AGPL-3.0
17	*YetiForceCRM*	YetiForceCompany	686	1523	72	PHP	YetiForce Non-commercial 6.5*
18	*inoERP*	inoerp	677	703	4	JavaScript	MPL-2.0
19	*stock-logistics-warehouse*	OCA	671	290	142	HTML	AGPL-3.0
20	*hr*	OCA	647	192	102	HTML	AGPL-3.0
21	*account-invoicing*	OCA	637	219	184	HTML	AGPL-3.0
22	*OpenUpgrade*	OCA	634	615	36	Python	AGPL-3.0
23	*product-attribute*	OCA	628	163	198	HTML	AGPL-3.0
24	*openeducat_erp*	openeducat	600	508	50	Python	LGPL-3.0
25	*account-financial-reporting*	OCA	587	201	126	Python	AGPL-3.0
26	*stock-logistics-workflow*	OCA	572	204	173	HTML	AGPL-3.0
27	*pos*	OCA	568	234	82	HTML	AGPL-3.0
28	*social*	OCA	554	134	179	HTML	AGPL-3.0
29	*metasfresh*	metasfresh	530	1,511	57	Java	GPL-2.0
30	*grocy*	grocy	497	5,740	81	Blade	MIT
31	*contract*	OCA	496	152	91	Python	AGPL-3.0
32	*bank-payment*	OCA	494	175	132	Python	AGPL-3.0
33	*l10n-spain*	OCA	490	204	95	Python	AGPL-3.0
34	*e-commerce*	OCA	463	148	89	HTML	AGPL-3.0
35	*server-ux*	OCA	462	128	156	HTML	AGPL-3.0
36	*manufacture*	OCA	447	149	107	Python	AGPL-3.0
37	*ofbiz-framework*	apache	441	612	67	Java	Apache-2.0
38	*ever-gauzy*	ever-co	420	1,577	75	TypeScript	various licenses
39	*WebVella-ERP*	WebVella	410	1,039	10	C#	Apache-2.0
40	*mixerp*	thcristo	401	7	1	JavaScript	MPL-2.0

*not in SPDX licenses list

Groovy, .NET, that appear in later positions in our dataset. C++ is number 14, Scala number 49, Kotlin number 29, Groovy number 38, whereas Visual Basic .NET appears in position 30 and ASP.NET in position 36. Languages that are now generally less popular are also found among the top ones, e.g., Pascal is number 15 in top languages overall according to the TIOBE[8] index.

[8] https://www.tiobe.com/tiobe-index/.

Table 3 shows also the average forks count, the average stars/watchers count per programming language, and the average value for days between the creation of the repository and the last update performed. The average number of forks and stars/watchers is relatively low in all languages. Forks are much more for repositories with Python as main programming language that may be attributed to the overall popularity of Python as programming language (it is now the most popular language according to TIOBE index). This also shows that repositories in Python may be reused more [22]. Stars/watchers are higher for the case of the Blade language, a general-purpose programming language focused on enterprise Web, Internet of Things and secure application development, that is the main programming language of 74 repositories in the dataset including the *Grocy*[9] ERP that is among the popular ones (number 30). Regarding the activity period (i.e., period of commits in the repository), repositories with the Ruby language are active for a much longer period than all other top languages. Repositories having as main programming language one of the top 6 languages have an average activity period in the range of 400–500 days.

Table 3. Top 20 programming languages in open source ERP systems.

	Main language	# repositories	% repositories	Avg. # forks	Avg. # stars/ watchers	# active days
1	JavaScript	1,636	14.23%	5.00	7.41	414.25
2	Python	1,492	12.97%	**28.69**	38.80	500.54
3	PHP	1,218	10.59%	7.18	15.09	540.30
4	Java	1,144	9.95%	4.74	9.42	407.79
5	HTML	795	6.91%	17.83	7.30	440.65
6	C#	777	6.76%	2.33	4.61	420.97
7	TypeScript	396	3.44%	1.60	5.70	247.41
8	CSS	355	3.09%	0.61	0.72	292.61
9	Ruby	162	1.41%	5.85	9.13	**1,130.66**
10	Vue	121	1.05%	1.81	4.79	280.64
11	Shell	104	0.90%	3.12	3.84	538.12
12	Jupyter Notebook	84	0.73%	0.44	1.63	268.95
13	Pascal	80	0.70%	2.75	2.35	574.85
14	C++	78	0.68%	1.23	3.68	541.65
15	MATLAB	76	0.66%	2.38	6.32	757.55
16	Blade	74	0.64%	8.72	**81.69**	287.45
17	Dart	70	0.61%	0.73	1.33	210.27
18	Go	67	0.58%	2.94	8.85	543.66
19	C	45	0.39%	5.38	8.77	625.02
20	TSQL	42	0.37%	2.33	3.5	406.05

Regarding licensing (using information from the *license.name* field returned by GitHub Search API), the vast majority of repositories does not indicate any

[9] https://grocy.info/.

license: 71.63% (8,238 repositories) do not indicate a license. The rest of the analysis for this RQ is based on the licensed repositories in the dataset, although we noted cases where GitHub did not return a license even though one exists (e.g., *scm-biz-suite* has an Apache-2.0 license according to its README file that was manually added in Table 2, as GitHub did not return this information, and *metasfresh* has a GPL-2.0 license that was similarly manually added).

The 26 different licenses that appear in the licensed repositories of the dataset (and also the case of *Other* license indicated in the GitHub API call result, since for 445 or 3.87% of repositories, NOASSERTION is returned for the license) are listed in Table 4, along with the number of repositories they appear in and the relevant percentages. The permissive MIT license is the top one among the repositories, indicating that most repository creators choose to put less restrictions on the software usage, distribution and reuse. It is followed by the strong copyleft license GPL-3.0 (GNU General Public License v3.0), the permissive Apache-2.0 (Apache License 2.0) and the strong copyleft AGPL-3.0 (GNU Affero General Public License v3.0). GPL and Apache licenses were also found among the top ones used in GitHub repositories in a previous work that analyzes repositories on license usage [21]. MIT was found fourth in the list in this previous work. Another work that analyzed Stack Exchange questions referring to licenses found as top 5 licenses general references to GPL, and also MIT, GPL-3.0, GPL-2.0 and Apache-2.0 (5th position) [16]. There is also a drop in the number of repositories a license appears in after the 5th top license. In the area of ERP systems, we can therefore say that licensing schemes that provide more freedom to the users of the software in terms of software reuse and distribution are more common. We observe also that some content licenses of Creative Commons that are not usually used for source code appear among the licenses: CC0-1.0, CC-BY-4.0, CC-BY-SA-4.0.

We examined the evolution of licenses over the years, i.e., if the licenses chosen by repositories change over the years. Considering again only repositories that include a license, Fig. 3 depicts the different licenses each year. To keep the figure readable, the top 10 licenses are only shown in the figure (and the indication of *Other* license). We see that the top license, MIT, is applied in an increasing number of repositories in recent years, especially since 2019. GPL-3.0 license has an approximately steady presence over the years, and the same applies to Apache-2.0 but with a decrease in 2023, considering that the number of new repositories is larger that year. So there is a decrease in the popularity of Apache-2.0 in open source ERP systems. LGPL-3.0 has a small presence over the years, although the most popular ERP system (i.e., *Odoo*) employs this license (in earlier years *Odoo* was using the AGPL-3.0 license).

Table 4. Licenses used in open source ERP systems.

	License (SPDX name)	License full name	# repositories	% repositories
1	MIT	MIT License	1,260	10.96%
2	GPL-3.0	GNU General Public License v3.0	634	5.51%
3	Apache-2.0	Apache License 2.0	364	3.17%
4	AGPL-3.0	GNU Affero General Public License v3.0	241	2.10%
5	GPL-2.0	GNU General Public License v2.0	107	0.93%
6	LGPL-3.0	GNU Lesser General Public License v3.0	41	0.36%
7	BSD-3-Clause	BSD 3-Clause "New" or "Revised" License	37	0.32%
8	Unlicense	The Unlicense	31	0.27%
9	BSD-2-Clause	BSD 2-Clause "Simplified" License	19	0.17%
10	CC0-1.0	Creative Commons Zero v1.0 Universal	19	0.17%
11	MPL-2.0	Mozilla Public License 2.0	11	0.10%
12	LGPL-2.1	GNU Lesser General Public License v2.1	10	0.09%
13	CC-BY-4.0	Creative Commons Attribution 4.0 International	8	0.07%
14	EPL-2.0	Eclipse Public License 2.0	5	0.04%
15	OSL-3.0	Open Software License 3.0	5	0.04%
16	ISC	ISC License	4	0.03%
17	WTFPL	Do What The F*ck You Want To Public License	3	0.03%
18	0BSD	BSD Zero Clause License	2	0.02%
19	BSD-3-Clause-Clear	BSD 3-Clause Clear License	2	0.02%
20	BSL-1.0	Boost Software License 1.0	2	0.02%
21	EPL-1.0	Eclipse Public License 1.0	2	0.02%
22	CC-BY-SA-4.0	Creative Commons Attribution Share Alike 4.0 International	2	0.02%
23	EUPL-1.1	European Union Public License 1.1	2	0.02%
24	UPL-1.0	Universal Permissive License v1.0	2	0.02%
25	Artistic-2.0	Artistic License 2.0	1	0.01%
26	BSD-4-Clause	BSD 4-Clause "Original" or "Old" License	1	0.01%
27	-	Other	445	3.87%

4.4 RQ4. Repositories Building upon Other Repositories

Since we observed that many repositories are add-ons to other main repositories (mainly *Odoo* ERP), in this RQ we are examining this in more detail. We are relying on the analysis of the description text of repositories in the dataset, in order to see whether there are references to other repositories of the dataset, using a regular expression to match word boundaries, so that exact words are only matched. Due to the large number of repositories in the dataset, we limit the search for mentions to other repositories to the repositories that have at least 10 forks (480 repositories in the dataset). Some repositories have the same name but come from different organizations (e.g., 593 repositories in the whole dataset have the name *erp* or *ERP*), so we removed duplicates from these names and used the remaining 460 distinct repository names. Table 5 shows the frequency of appearances of repository names in other repositories of the dataset. We removed common repository names that do not assist in understanding repository reuse, since they do not point necessarily to another repository but to terminology commonly used: erp, project, web, crm, pos (Point of Sale), rma (Return Merchandise Authorization), edi (Electronic Data Interchange), hr (Human Resources), docker, shop, e-commerce, payroll, point, workflow, social,

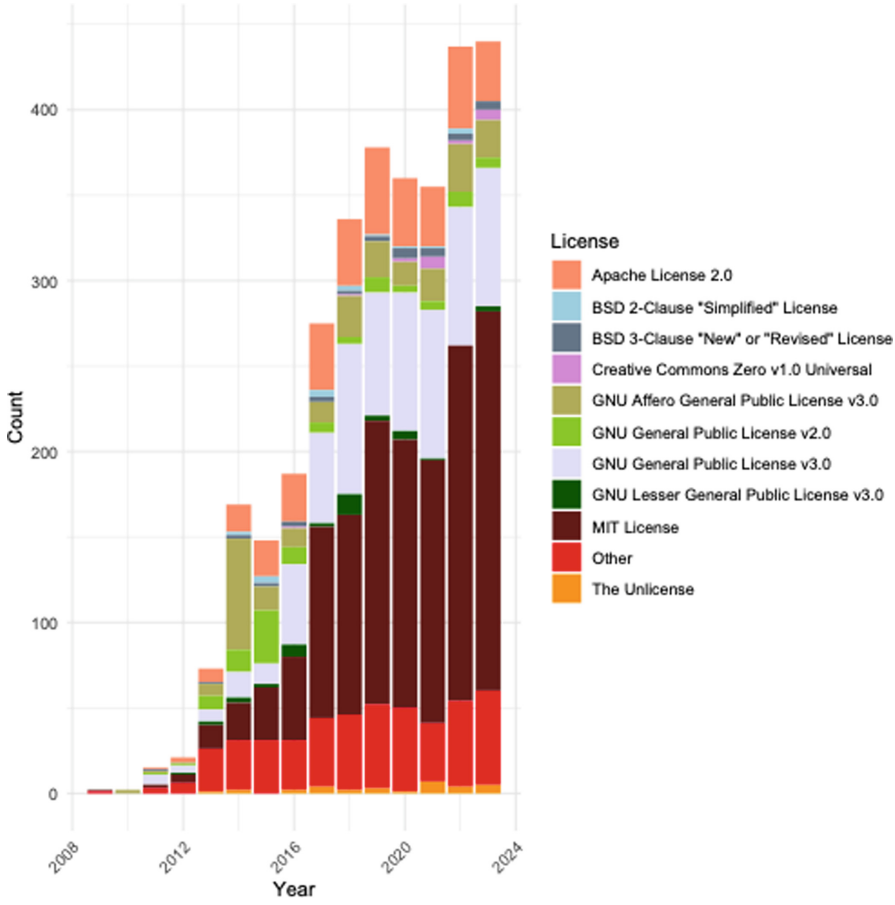

Fig. 3. License distribution over the years for the top 10 licenses.

health, timesheet, manufacture, currency, contract, helpdesk, cooperative, commission, wms (Warehouse Management System), knowledge, brand, ones, fleet, sister.

From the table, we observe that the top repositories in the dataset are reused or extended in other repositories with *Odoo* and *ERPNext* being the ones with the largest community (they are also the top ERPs in terms of popularity from the results of RQ2). Below are some examples of reuse cases from the results of Table 5:

- *Escodoo/fleet-addons* includes modules that extend *Odoo*: "*Escodoo add-ons used to extend or customize Odoo Fleet app functions.*"
- *libracore/erpnext* extends *ERPNext*, indicating in its description that it is "*the libracore flavour of ERPNext. It is derived from the original frappe/erpnext source, but enriched and tested.*"

- *keshavgaddhyan/ERPNext* according to its description is an "*AI Analysis for erpnext Sales Data.*"
- *devcoffee/idempiere-pt_br* is the translation of *idempiere* to Portuguese: "*pt_BR translation project for iDempiere Open Source ERP.*"
- *lynnaloo/xtuple-nodejs-rest-client-example* is according to its description a "*xTuple ERP Node.js REST API Client Example using the Google APIs Client Library for Node.js*", so it relies on *xTuple.*

Table 5. Repository names present in other repositories' descriptions.

Repository name	Frequency	Repository name	Frequency
odoo	472	*erplab*	3
erpnext	316	*growerp*	3
dolibarr	55	*ekylibre*	2
idempiere	37	*trytond*	2
tryton	22	*uzerp*	2
xtuple	14	*phreebooks*	2
adempiere	12	*metasfresh*	1
grocy	11	*erpjs*	1
pdv	10	*stoq*	1
openbravo	9	*open-erp*	1
hexya	6	*koalixcrm*	1
netsuite	6	*bookyt*	1
frontaccounting	6	*yycoin*	1
flectra	4		

5 Take Away Messages

Besides the discussion within the answers to our RQs, we point out the following take away messages of this work:

ERP Systems Features: popular OSS ERP system repositories integrate different business functions, e.g., *ERPNext* covers more than 14 business areas, are build on popular frameworks, e.g., *akaunting* uses Laravel, and have a readable, nicely structured and extended README file (with separate sections, text highlighting, figures). So we argue that the technologies choice and the extended features are necessary for a successful OSS ERP system in terms of use and reuse. Less popular repositories may need to invest more on those aspects. Some repositories also have an impressive number of contributors (e.g., *dolibarr* in Table 2), so maintaining a permissive open source model may be beneficial for repositories currently having a small number of repositories. It is not a necessity however, as some manage to gain popularity with a lower contributors' number (e.g., *scm-biz-suite* in Table 2).

Licensing: the vast majority of repositories does not indicate any license (using information from the *license.name* field returned by GitHub Search API): 71.63% (8,238 repositories) do not indicate a license. GitHub allows the creation of a repository without a license as the respective field is optional. This is a major issue, as repositories with no license indication cannot be reused, even though they are public (there is no legal permission to reuse the code).[10] It is a major issue also in the case of open source ERP systems indicating also that the largest percentage of the repositories cannot be reused legally. They might be reused but only on a private way and cannot be distributed. Nevertheless, such repositories have also been used to train Large language models (LLMs), such as GitHub Copilot. So actually the number of repositories that can be reused is smaller than the whole set of repositories in the dataset. As the above information is based on the *license.name* field returned by GitHub Search API, we noted cases where GitHub did not provide the repository license, even though a license existed. The real number of unlicensed repositories may thus, be lower but still the percentage is high, even considering this inaccuracy. Manual analysis on the repositories in future work could verify this finding.

Communities Around Specific Tools: Some tools and primarily *Odoo* and *ERPNext* manage to maintain an extended community around them, with a large number of add-ons and extensions in separate repositories. This shows an extended reuse around specific core tools, and may be linked with the licensing scheme of these tools, e.g., *Odoo* has an LGPL-3.0 license, but its add-ons are licensed under AGPL-3.0. Overall, a small number of OSS ERPs are reused to a large extent, indicating that although a very large number of OSS ERPs exist, few have managed to secure their presence: in out dataset, 9 repositories are indicated in at least 10 other repositories' descriptions according to the results of RQ4.

6 Limitations

Construct Validity. Our data collection process relied on two keywords to collect repositories and our manual analysis on a repositories sample has found a small number of irrelevant repositories that were removed. We performed filtering on the repository topics and description to filter out such cases but this process may have excluded also relevant repositories that did not meet the filtering criteria.

External Validity. As we rely on GitHub for the data collection, our results may not apply to open source ERPs hosted in other locations (e.g., GitLab, Bitbucket). We argue that GitHub is very popular among practitioners and researchers and we expect that the vast majority of open source ERPs are hosted on GitHub. This is why it was selected as data source in this work.

[10] https://choosealicense.com/no-permission/.

Internal Validity. We are relying on existing tools provided by GitHub, since these tools are used to provide the data returned by the GitHub Search API. For the case of license indication, we observed a number of cases where no license is indicated in the GitHub Search API response, although the repository indicates a license, meaning that GitHub cannot detect the license correctly in some cases and this may have affected our results concerning license usage.

Other Limitations. We have not focused our analysis on specific types of ERP systems, although many exist. Therefore, our results may vary if we consider specific business operations supported by an ERP system (e.g., supply chain management, human capital management).

7 Conclusions

In this work, we have presented the main characteristics and the most popular open source ERP systems hosted on GitHub, using a dataset of 11,500 repositories. The results of this work can help in understanding the community of open source ERPs, some of its reuse practices and may serve as comparison point with commercial ERPs. Organizations that intend to adopt an open source ERP system or extend it can also use the results to select the appropriate system for reuse based their needs (e.g., based on popularity using number of forks or the software license used). But primarily, it can serve as a first step towards a framework that is more specific on modules' reuse within OSS ERPs. As future work, we would like to work on this aspect and analyze the source code of the open source ERPs to identify common libraries reused among repositories and security/privacy mechanisms applied. Manual analysis on the repositories would also be useful for better understanding usage of OSS licenses. We finally intend to expand the analysis to include other types of information systems that are explicitly indicated as such, such as CRMs.

References

1. Bjelland, E., Haddara, M.: Evolution of ERP systems in the cloud: a study on system updates. Systems **6**(2), 22 (2018)
2. Borges, H., Hora, A., Valente, M.T.: Understanding the factors that impact the popularity of GitHUB repositories. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 334–344. IEEE (2016)
3. Businge, J., Openja, M., Nadi, S., Berger, T.: Reuse and maintenance practices among divergent forks in three software ecosystems. Empir. Softw. Eng. **27**(2), 54 (2022)
4. Constantinou, E., Kapitsaki, G.M.: Identifying developers' expertise in social coding platforms. In: 2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 63–67. IEEE (2016)
5. Cosentino, V., Izquierdo, J.L.C., Cabot, J.: A systematic mapping study of software development with GitHUB. IEEE access **5**, 7173–7192 (2017)
6. Foukarakis, I.E., Kapitsaki, G.M., Tselikas, N.D.: Choosing licenses in free open source software. In: SEKE, pp. 200–204 (2012)

7. Jazayeri, B., Schwichtenberg, S., Küster, J., Zimmermann, O., Engels, G.: Modeling and analyzing architectural diversity of open platforms. In: Dustdar, S., Yu, E., Salinesi, C., Rieu, D., Pant, V. (eds.) CAiSE 2020. LNCS, vol. 12127, pp. 36–53. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-49435-3_3
8. Juell-Skielse, G., Nilsson, A.G., Nordqvist, A., Westergren, M.: Ames: towards an agile method for ERP selection. In: CAISE Forum 2012, Gdańsk, Poland, 28 June 2012, pp. 98–105 (2012)
9. Kirikova, M.: Challenges in enterprise and information systems modeling in the contexts of socio cyber physical systems. In: Pergl, R., Babkin, E., Lock, R., Malyzhenkov, P., Merunka, V. (eds.) EOMAS 2019. LNBIP, vol. 366, pp. 60–69. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-35646-0_5
10. Kochanthara, S., Dajsuren, Y., Cleophas, L., van den Brand, M.: Painting the landscape of automotive software in GitHUB. In: Proceedings of the 19th International Conference on Mining Software Repositories, pp. 215–226 (2022)
11. Mladenova, T.: Open-source ERP systems: an overview. In: 2020 International Conference Automatics and Informatics (ICAI), pp. 1–6. IEEE (2020)
12. Mukelabai, M., Derks, C., Krüger, J., Berger, T.: To share, or not to share: exploring test-case reusability in fork ecosystems. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 837–849. IEEE (2023)
13. Oliveira, J., Fernandes, E., Souza, M., Figueiredo, E.: A method based on naming similarity to identify reuse opportunities. In: Anais do Simpósio Brasileiro de Sistemas de Informação (SBSI). SBSI 2016, Sociedade Brasileira de Computação (2016). https://doi.org/10.5753/sbsi.2016.5976
14. Oliveira, J., Fernandes, E., Vale, G., Figueiredo, E.: Identification and prioritization of reuse opportunities with JReuse. In: Botterweck, G., Werner, C. (eds.) ICSR 2017. LNCS, vol. 10221, pp. 184–191. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56856-0_13
15. Olson, D.L., Staley, J.: Case study of open-source enterprise resource planning implementation in a small business. Enterp. Inf. Syst. **6**(1), 79–94 (2012)
16. Papoutsoglou, M., Kapitsaki, G.M., German, D., Angelis, L.: An analysis of open source software licensing questions in stack exchange sites. J. Syst. Softw. **183**, 111113 (2022)
17. Rashid, M.A., Hossain, L., Patrick, J.D.: The evolution of ERP systems: a historical perspective. In: Enterprise Resource Planning: Solutions and Management, pp. 35–50. IGI global (2002)
18. Rigas, E.S., Papoutsoglou, M., Kapitsaki, G.M., Bassiliades, N.: Mining software repositories to identify electric vehicle trends: the case of GitHUB. In: 2023 10th International Conference on Behavioural and Social Computing (BESC), pp. 1–6 (2023). https://doi.org/10.1109/BESC59560.2023.10386526
19. Russell, P.H., Johnson, R.L., Ananthan, S., Harnke, B., Carlson, N.E.: A large-scale analysis of bioinformatics code on GitHUB. PLoS ONE **13**(10), e0205898 (2018)
20. Sheoran, J., Blincoe, K., Kalliamvakou, E., Damian, D., Ell, J.: Understanding "watchers" on GitHUB. In: Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 336–339 (2014)
21. Vendome, C., Bavota, G., Penta, M.D., Linares-Vásquez, M., German, D., Poshyvanyk, D.: License usage and changes: a large-scale study on GitHUB. Empir. Softw. Eng. **22**, 1537–1577 (2017)

22. Xu, B., An, L., Thung, F., Khomh, F., Lo, D.: Why reinventing the wheels? An empirical study on library reuse and re-implementation. Empir. Softw. Eng. **25**, 755–789 (2020)
23. Zhou, S., Vasilescu, B., Kästner, C.: What the fork: a study of inefficient and efficient forking practices in social coding. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 350–361 (2019)

Author Index

A. Achilleos et al. (Eds.): ICSR 2024, LNCS 14614, p. 189, 2024.
https://doi.org/10.1007/978-3-031-66459-5