On Preserving the Behavior in Software Refactoring: A Systematic Mapping Study

Eman Abdullah AlOmar[a,*], Mohamed Wiem Mkaouer[a], Christian Newman[a], Ali Ouni[b]

[a] *Rochester Institute of Technology, Rochester, NY, USA*
[b] *ETS Montreal, University of Quebec, Montreal, QC, Canada*

Abstract

Context: Refactoring is the art of modifying the design of a system without altering its behavior. The idea is to reorganize variables, classes and methods to facilitate their future adaptations and comprehension. As the concept of behavior preservation is fundamental for refactoring, several studies, using formal verification, language transformation and dynamic analysis, have been proposed to monitor the execution of refactoring operations and their impact on the program semantics. However, there is no existing study that examines the available behavior preservation strategies for each refactoring operation.
Objective: This paper identifies behavior preservation approaches in the research literature.
Method: We conduct, in this paper, a systematic mapping study, to capture all existing behavior preservation approaches that we classify based on several criteria including their methodology, applicability, and their degree of automation.
Results: The results indicate that several behavior preservation approaches have been proposed in the literature. The approaches vary between using formalisms and techniques, developing automatic refactoring safety tools, and performing a manual analysis of the source code.
Conclusion: Our taxonomy reveals that there exist some types of refactoring operations whose behavior preservation is under-researched. Our classification also indicates that several possible strategies can be combined to better detect any violation of the program semantics.

Keywords: refactoring, behavior preservation, systematic mapping study

*Corresponding author
Email addresses: eman.alomar@mail.rit.edu (Eman Abdullah AlOmar), mwmvse@rit.edu (Mohamed Wiem Mkaouer), cdnvse@rit.edu (Christian Newman), ali.ouni@etsmtl.ca (Ali Ouni)

1. Introduction

Software maintenance and evolution is an essential activity for any software system and the success of a system measures by its ability to maintain a high quality of design in the face of continuous changes. Because the change to the code base is inevitable, mechanisms must be employed in order to avoid causing deterioration to its integrity. One of the key mechanisms to cope with this challenge is refactoring. Refactoring is the process of optimizing the internal structure of the code without changing its external behavior. With the existence of many refactoring techniques, developers are still reluctant to rely on refactoring frameworks, and they prefer to refactor their code manually [1, 2]. Surveys have revealed developer's lack of trust in automatic refactoring [3], due to the fear of breaking the code semantics and introducing bugs. Although refactoring, by definition, guarantees the safety and preservation of the refactored system's functionality, its adoption is still limited. One way to narrow the gap between refactoring and its adoption, is to highlight the existing effort in securing the execution of refactoring operations. However, little is known about how existing verification techniques allow such variety of changes, which vary from renaming methods and attributes, to extracting classes and merging packages, to be executed without altering the software's functionality. Thus, There is a lack of comprehensive studies to keep researchers and practitioners up-to-date with the status of research in preserving the behavior, evaluating the correctness of the transformation, and whether or not these approaches lead to a safe and trustworthy refactoring.

While refactoring has been the focus on several SLRs, these studies have mainly focused on identifying refactoring opportunities, through the identification of code smells, as a detection step, and on recommending the appropriate refactoring operations, as a correction step. Our work is different from these papers since our SLM primarily focuses on collecting and summarizing all the behavior preservation techniques in all areas of software refactoring. It is not limited to design-based approaches; it also covers code-based behavior preservation approaches. To the best of our knowledge, no previous work has conducted a comprehensive SLM pertaining to behavior preservation techniques in software refactoring.

The goal of this paper is to report an SLM that (1) identifies behavior preservation approaches in the research literature, and (2) identifies open issues in existing research. The outcomes of this SLM can serve as summarizing indexes and are expected to (1) assist researchers to identify related behavior preservation topics that are not well explored, and (2) guide practitioners to know the existing techniques for behavior preservation, which have an impact on refactoring decisions made in practice.

To conduct this systematic mapping study, we followed established guidelines for SLR and SLM studies in SE [4, 5, 6]. We performed the review by defining the search string, the search academic article search engine, the selection criteria, and the research questions. We extracted data for 101 potentially relevant articles using the search academic article search engine. After careful

screening of these articles, we identified 28 primary studies (PSs). We classified these PSs based on different perspectives, including the software artifacts and language paradigms, the refactoring operations, the behavior preservation approaches, and the evaluation methods considered. We identified several topics and challenges in need to be addressed in future research.

2. Background & Related Work

2.1. Behavior preserving transformation

Refactoring is a maintenance task in which the internal structure of the source code is improved while the external behavior is preserved [7]. The definition of behavior preservation, originally introduced by Opdyke [8], states that, *for the same set of input values, the resulting set of output values should be the same before and after the refactoring.* Opdyke supports the notion of behavior preservation by specifying refactoring preconditions. An example of a refactoring precondition can be seen when considering *Extract Class* refactoring in which naming conflicts must be avoided. Opdyke defined seven properties that must be checked before refactoring programs, which include: (1) unique superclass, (2) distinct class names, (3) distinct member names, (4) inherited member variables not redefined, (5) compatible signatures in member function redefinition, (6) type-safe assignment, and (7) semantically equivalent references and operations.

Some refactoring techniques and formalisms to guarantee program preservation have been reported in a survey study by Mens and Tourwe [9]. They discussed the existing literature in terms of refactoring activities applied and their techniques, the application of refactoring to any type of software artifacts, refactoring tool support, and the impact of refactoring on the software process. In one of these several refactoring classification aspects, they discussed how the use of assertions (preconditions, postconditions, and invariants) and the use of graph transformation could help in guaranteeing behavior preservation. Therefore, in contrast to this SLM, the previous survey does not cover all of the approaches to guarantee behavior-preserving transformation. The survey considered only a few studies on behavior preservation because of its broader topic in the area of software refactoring and because it was performed a decade ago.

2.2. Other systematic literature reviews in refactoring

This work is a systematic mapping study in which we studied and summarized the primary studies (PSs) reporting the behavior preservation approach in the area of software refactoring. We did not find any SLM discussing the behavior preservation strategies. However, we reviewed a number of existing SLRs because of the similarities between those works and ours in terms of research setting. Table 1 summarizes the SLRs cited in this study.

Zhang et al. [10] conducted an SLR of 39 studies in the field of bad code smells. They discussed these studies based on the following: the goals of the

Table 1: Refactoring-related SLRs in Related Work.

Study	Year	Focus	No. of PSs
Zhang et al. [10]	2011	Bad smells & refactoring	39
Abebe and Yoo [11]	2014	Trends, opportunities & challenges of software refactoring	58
Misbhauddin and Alshayeb [12]	2015	UML model refactoring	94
AlDallal [13]	2015	Refactoring opportunities identification	47
Singh and Kaur [14]	2017	Refactoring opportunities identification	238
AlDallal and Abdin [15]	2017	Impact of refactoring on quality	76
Mariani and Vergilio [16]	2017	Search-based refactoring	71
Baqais and Alshayeb [17]	2020	Automatic refactoring	41

studies, type of code smells addressed, the approaches to studying code smells, and identifying bad smells and refactoring opportunities.

In a systematic review reported by Abebe and Yoo [11], 58 studies were reviewed with the intention of revealing the trends, opportunities, and challenges of software refactoring. Their classification helped guide researchers to address the crucial issues in the field of software refactoring.

Misbhauddin and Alshayeb [12] performed an SLR in the area of refactoring UML models. They analyzed and classified 94 PSs based on several criteria: UML types of models, the formalisms used, and the effect of refactoring on model quality. In part of the research, they listed a few model behavior specification approaches. Our SLM is not limited to design-based approaches; it also covers code-based behavior preservation approaches.

AlDallal [13] conducted an SLR of 47 PSs published on identifying refactoring opportunities in object-oriented code. AlDallal's review classified PSs based on the considered refactoring scenarios, the approaches to determine refactoring candidates, and the datasets used in the existing empirical studies. In a following SLR work by AlDallal and Abdin [15], they discussed 76 PSs and classified based on refactoring quality attributes of object-oriented code.

Singh and Kaur [14] performed an SLR as an extension of AlDallal's [13] SLR. In their review, they analyzed 238 research items in the field of code smell detection and its refactoring opportunities with the intention of addressing some research questions that were left open in AlDallal's SLR.

More recently, Baqais and Alshayeb [17] conducted a systematic literature review on automated software refactoring. In their review, they analyzed 41 studies that propose or develop different automatic refactoring approaches.

In the area of search-based refactoring, Mariani and Vergilio [16] systematically reviewed 71 studies and classified them based on the main elements of search-based refactoring, including artifacts used, encoding and algorithms used, search technique, metrics addressed, available tools, and conducted evaluation. Within the field of search-based refactoring, Mariani and Vergilio classified the selected PSs into five general categories related to behavior preservation methods. These categories involved: (1) Opdyke's function [8], (2) Cinnéide's function [18], (3) domain-specific, (4) no evidence of behavior preservation, and (5) do not mention the method. The current SLM does not overlap Mariani and Vergilio's SLR because this SLM entirely focuses on behavior preservation trans-

formation in all areas of software refactoring, whereas Mariani and Vergilio's SLR mainly focused on search-based refactoring and discussed partially general behavior preservation methods.

As shown in Table 1, all the above-mentioned studies focus on either (1) detecting refactoring opportunities, through the optimization of structural metrics, or the identification of design and code defects, or (2) automating the generation and recommendation of the most optimal set of refactorings to improve the system's design while minimizing the refactoring effort, so that developers still can recognize their own design. Our work is different from these papers since our SLM primarily focuses on collecting and summarizing all of the behavior preservation techniques in all areas of software refactoring. It is not limited to design-based approaches; it also covers code-based behavior preservation approaches. To the best of our knowledge, no previous work has conducted a comprehensive SLM pertaining behavior preservation techniques in software refactoring.

3. Research Method

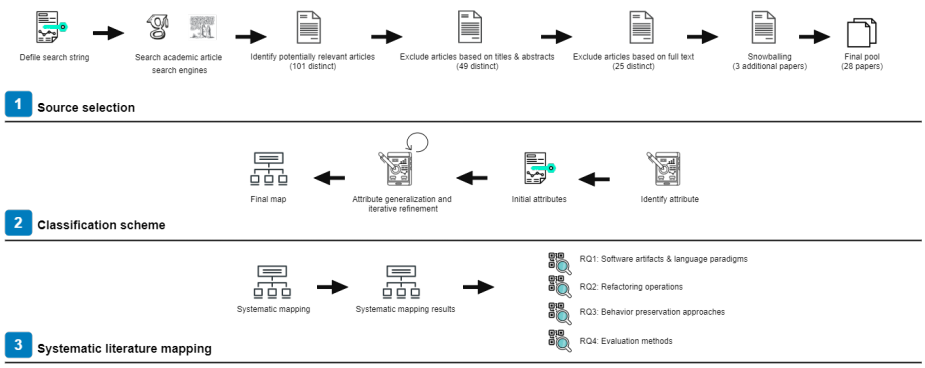


Figure 1: Literature Search Process.

This SLM aggregates and summarizes the approaches in the field of testing behavior preservation in software refactoring. Based on the established guidelines [4, 5, 6], we performed the SLM in three main phases: planning, conducting, and reporting the review. Creating a protocol is a major step when conducting an SLM [4]. This protocol contains the research questions, search strategy, study selection including inclusion and exclusion criteria, and data extraction and analysis to answer research questions.

The core motivation behind carrying out this SLM is to:

- Identify behavior preservation approaches in research literature.
- Identify open issues in existing research.

3.1. Research questions

Since little is known about the literature review of behavior preservation, this SLM serves as an exploration of this topic to extract existing techniques, currently being used, and their associated programming languages. The analysis of such wide variety of methods leads to develop a categorization and reveals areas of potential improvements. Therefore, we follow criteria defined in [4, 5, 6]when defining our research questions. The motivation behind each question is described below.

3.1.1. RQ1: What types of software artifacts and language paradigms were covered in the PSs to examine behavior preservation?

The first research question explores the types of system levels and their language paradigms considered in the PSs, and to know what software artifacts are mostly used in the literature.

3.1.2. RQ2: What refactoring types were considered in the PSs?

Research question two identifies the refactoring operations that are tested and evaluated by behavior-preserving transformation approaches. This RQ serves as a popularity context to reveal the most and least popular refactoring types. Yet, the popularity in the context of behavior preservation is an indicator for refactoring complexity, as a code transformation, and thus, it potential proneness to errors.

3.1.3. RQ3: What approaches were considered by the PSs to test the behavior-preserving transformations in software refactoring?

We pose this research question to study current approaches for testing behavior preservation of refactoring, and to get an overview of what different criteria are addressed by the existing methods. Accordingly, we collect information about refactoring techniques, automated analyses, and the manual analysis approach. Lastly, we check if the proposed approach is compared with existing methods, and study the pros and cons of the current approaches to suggest areas for improvement.

3.1.4. RQ4: What evaluation methods were used in the PSs to assess the proposed behavior preservation approaches?

We answer this research question by investigating how researchers evaluate and validate their proposed approaches in practice, when checking the reliability of the obtained conclusions. The answer to this question enumerates all evaluation methods that are found to be appropriate and most reliable when validating behavior preservation approaches.

3.2. Search strategy

To find relevant studies, we performed an automatic search in Google Scholar and Scopus [1]. These search engines cover all main venues (e.g., IEEE, ACM, Springer). Our search string in these search engines was:

> ((behavior-preserving OR behavior preserving OR behavior preservation OR behaviour-preserving OR behaviour preserving OR behaviour preservation OR preserv* behavior OR preserv* behaviour) AND (formal OR method OR approach) AND (refactor* OR restructur*))

TextBox 1: Search String.

The strategy to construct search keywords was as follows:

- Derive the main terms from research questions and terms considered in the relevant papers.
- Include alternative spellings for major terms.
- Combine possible synonyms and spellings of the main terms using the Boolean OR operators, and then combine the main terms using the Boolean AND operators.

These search keywords are applied to paper titles, abstracts, and keywords. To check the validity of the search string, we manually double check a few articles. Similar to [19], to restrict the search space when using Google Scholar to execute search string, we checked first several pages because we noticed that relevant studies appear in the first few pages. The process of determining the final list of PSs is depicted in Figure 1.

3.3. Study selection

To collect the PSs, we adapted the search process of [15] and conducted a four phased process.

3.3.1. Stage 1

In this first stage of the paper selection process, given in Figure 1, we searched the academic article search engine for potentially related articles. Our criteria included applying our predefined search string against a publication's title, abstract, and keyword fields. Results from this search were not limited to specific venues. Searching the Google Scholar and Scopus resulted in a total of 101 literature publications. To reduce the possibility of including totally irrelevant articles, we performed the initial screening of the articles. Literature

[1] www.scopus.com

publications were then eliminated based on the defined inclusion and exclusion criteria to filter our irrelevant articles gathered in Stage 1.

Inclusion criteria:

The selected studies must satisfy all the following inclusion criteria:

- The article must be published in a peer-reviewed journal or conference before March 1, 2021.
- The article must report an approach to testing behavior preservation and verify the correctness of refactorings.

Exclusion criteria:

Papers are excluded if satisfying any of the exclusion criteria, as follows:

- The study did not report an approach to test behavior-preserving transformations in software refactorings.
- The study is a positioning paper, abstract, editorial, keynote, tutorial, or panel discussions.
- The study is not written in English

Regarding the second inclusion criteria, we only considered PSs that reported an approach to test the behavior preservation in refactoring, so we excluded any other articles that provided broad explanation about the concept of behavior preservation. Additionally, we excluded articles that were short because of their lack of comprehensiveness, e.g., [20].

3.3.2. Stage 2

This stage involved an elimination of studies that were returned by the academic article search engine on the basis of the titles and abstracts of the potentially relevant articles. It is important to consider the abstracts in this stage because the titles of some articles could be misleading. The inclusion and exclusion rules were applied at this stage to all retrieved studies. This elimination process reduced our result set to 49 literature publications.

3.3.3. Stage 3

To obtain the relevant PSs, the complete literature publication was read and reviewed. Literature publications were eliminated based on the defined exclusion and inclusion rules. This process resulted in a total of 28 literature publications that were accepted for this study.

3.3.4. Stage 4

To maximize the search coverage of all relevant papers, we conducted the snowballing technique [5] on papers already in the pool. It resulted in adding 3 additional papers, increasing the pool size to 28.

3.4. Data extraction

In order to determine the attribute(s) of the classification dimension, we screened the full texts of the PSs and identified the attribute(s) of that dimension. We used attribute(s) generalization and refinement to derive the final map, similar to [19]. After the extraction of the classification dimension, we read the selected PSs in detail to answer the research questions. We then extracted the standard information from each paper, similar to [21], and included the additional attributes relevant to our study to the form. The data extraction form used is shown in Table 2.

Data stored in [F1] to [F11] are for documentation purposes, whereas data in [F12] to [F21] are for the purpose of data analysis. This form enables us to report the details needed for the PSs in this SLM.

Table 2: Data Extraction Form.

No.	Field	Additional comments
F1	Primary study ID	N/A
F2	Author(s)	N/A
F3	Title	N/A
F4	Source	N/A
F5	Keyword	N/A
F6	Publication venue	N/A
F7	Type of publication	N/A
F8	Date of publication	N/A
F9	Publication details for journal	N/A
F10	Citation count (Google Scholar)	N/A
F11	Page numbers	N/A
F12	Approach	Method used to ensure behavior preservation
F13	Approach Subcategory	A subcategory of each approach
F14	Strategy	A specific strategy used for that method to ensure behavior preservation
F15	Artifacts	System levels refactoring
F16	Language Paradigm	A classification of software artifacts based on their features (if available)
F17	Refactorings	List of refactoring scenarios
F18	Refactoring Classification	A classification for each refactoring operation
F19	Evaluation Methods	A method used to validate and evaluate the proposed approach
F20	Strength	A brief description of method's strengths
F21	Limitation	A brief description of method's limitations

4. Results

4.1. Overview of the PSs

The research method discussed in Section 3 resulted in 28 relevant PSs listed in Appendix A. The main venues for these relevant PSs are presented in Table 3. The PSs were published in 18 different sources including journals and conferences. The list includes eleven journals and eleven conferences. The first relevant article discusses an approach of behavior preservation published in a journal in 1997, whereas the most recent one was published in 2018. The number of literature publication published in journals and conferences individually and combined are presented in Figure 2.

Except for [22], all the authors of the PSs are from academia. The authors of [22] are from industry. This indicates that most of the studies in this area were performed within an academic environment.

Table 4 shows an overview of the most-cited articles which indicate the degree of the most impactful PSs.

Table 3: Publication Sources.

Study	Year	Venue	Source
Roberts et al. [23]	1997	Journal	Theory and Practice of Object Systems (TAPOS)
Mens et al. [24]	2003	Journal	Journal of Software Maintenance and Evolution (SME)
Tip et al. [22]	2003	Conference	Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)
Garrido and Meseguer [25]	2006	Conference	International Workshop on Source Code Analysis and Manipulation (SCAM)
Straeten et al. [26]	2007	Journal	Software and System Modeling (SSM)
Massoni et al. [27]	2008	Conference	Fundamental Approaches to Software Engineering (FASE)
Soares et al. [28]	2009	Conference	Brazilian Symposium on Software Engineering (SBES)
Ubayashi et al. [29]	2008	Conference	International Conference on Software Testing, Verification, and Validation (ICST)
Schäfer et al. [30]	2008	Conference	Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)
Soares et al. [31]	2009	Conference	Brazilian Symposium on Programming Languages (SBLP)
Tsantalis and Chatzigeorgiou [32]	2009	Journal	IEEE Transactions on Software Engineering (TSE)
Schäfer and Moor [33]	2010	Conference	Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)
Soares et al. [34]	2010	Journal	IEEE Software
Tsantalis and Chatzigeorgiou [35]	2010	Journal	Journal of Systems and Software (JSS)
Tip et al. [36]	2011	Journal	Transactions on Programming Languages and Systems (TOPLAS)
Soares et al. [37]	2011	Conference	Brazilian Symposium on Programming Languages (SBLP)
Overbey and Johnson [38]	2011	Conference	International Conference on Automated Software Engineering (ASE)
Soares et al. [39]	2011	Conference	International Conference on Software Maintenance and Evolution (ICSME)
Soares et al. [40]	2013	Journal	Journal of Systems and Software (JSS)
Jonge and Visser [41]	2012	Conference	Workshop on Language Description (WLD)
Noguera et al. [42]	2012	Conference	International Conference on Software Maintenance and Evolution (ICSME)
Thies and Bodden [43]	2012	Conference	International Symposium on Software Testing and Analysis (ISSTA)
Mongiovi et al. [44]	2014	Journal	Science of Computer Programming (SCP)
Najaf et al. [45]	2016	Journal	Computing and Informatics (CI)
Horpácsi et al. [46]	2017	Conference	Verification and Program Transformation (VPT)
Mongiovi et al. [47]	2017	Journal	IEEE Transactions on Software Engineering (TSE)
Chen et al. [48]	2018	Journal	Information and Software Technology (IST)
Insa et al. [49]	2018	Journal	Scientific Programming (SP)

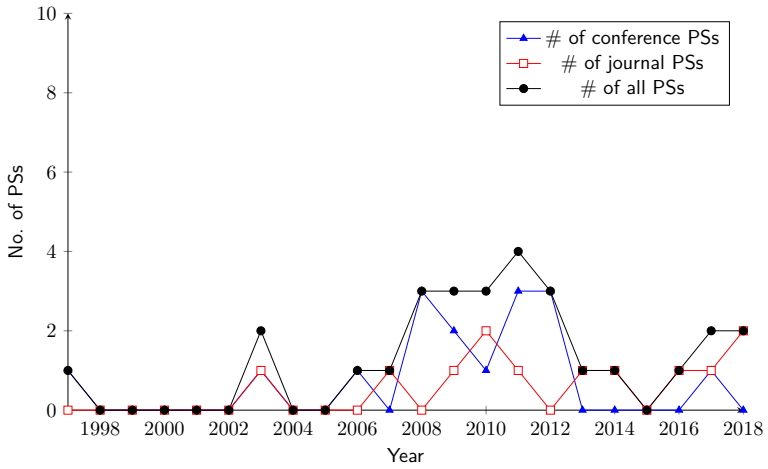


Figure 2: Distribution of Primary Studies by Year.

4.2. RQ1: What types of software artifacts and language paradigms were covered in the PSs to examine behavior preservation?

Table 5 presents the types of software artifacts, different language paradigms and programming and modeling languages used in the PSs. Refactoring is applied to only two kinds of artifacts in the literature: code and model. Code

Table 4: Citation Count (Obtained from Google Scholar).

Study	Year	Source	Count
Roberts et al. [23]	1997	TAPOS	550
Tsantalis & Chatzigeorgiou [32]	2009	TSE	316
Mens et al. [24]	2003	SME	206
Tip et al. [22]	2003	OOPSLA	181
Soares et al. [34]	2010	IEEE Software	122
Schäfer & Moor [33]	2010	OOPSLA	104
Schäfer et al. [30]	2008	OOPSLA	103
Tip et al. [36]	2011	Journal	74
Straeten et al. [26]	2007	SSM	64
Tsantalis & Chatzigeorgiou [35]	2010	JSS	59
Garrido & Meseguer [25]	2006	SCAM	49
Soares et al. [40]	2013	JSS	37
Overbey & Johnson [38]	2011	ASE	33
Mongiovi et al. [44]	2014	SCP	31
Soares et al. [39]	2011	ICSME	29
Thies and Bodden [43]	2012	ISSTA	23
Massoni et al. [27]	2008	FASE	21

refactoring targets to apply refactoring techniques at the source code level. Model refactoring aims to apply refactorings at model level as opposed to the source code. Most (82.75%) of the PSs were about refactored source code, and a few (17.24%) concerning refactored design models. Articles optimizing code are primarily focused on Java programming language. Few articles, however, used C++, Smalltalk, AspectJ, Fortran, PHP, BC, Erlang, Stratego, Mobl, and XML to test behavior preservation. For model refactoring, research deals with Alloy specification language or UML models. As can be seen from the table, most of the papers consider refactoring source code, focusing primarily on the Java language. Model refactoring is being used by few articles. Moreover, one of the articles [24] does not explicitly mention what types of artifacts were refactored. By analyzing the PS [24], it is possible to guess that it is applicable to either code or models since the behavior preservation approach described is about graph transformation.

The focus on Java language might be because of the popularity of Java, and refactoring examples in Fowler's book are written in Java. Researchers are encouraged to focus on different languages and apply more refactoring to design models when testing behavior preservation.

Table 5: Software Artifacts and its Language Paradigms.

Software Artifact	Language Paradigm	Language	PSs
Code	Class-based OO	Java	[44][28][31][34][32][25][22][39][47][40] [48] [43] [35] [41] [42] [36] [33] [30] [36]
		C++	[32]
		Smalltalk	[23]
	Aspect-oriented	AspectJ	[44] [37] [29] [42] [30]
	Imperative	Fortran, PHP, BC	[38]
	Functional	Erlang	[46] [49]
	Domain-specific	Stratego, Mobl	[41]
	Markup	XML	[42]
Model	Structural & Behavioral	UML	[26] [45]
	Structural formal	Alloy	[27]

Summary. Refactoring studies cover mainly two levels of artifacts: Source code and model. Source code artifacts are the main focus of refactoring literature. Java is the most popular programming language in these studies.

4.3. RQ2: What refactoring types were considered in the PSs?

As shown in Table 6 and Table 7, the literature publications addressed 150 distinct refactoring operations. In this SLM, we classify refactoring operations considered in the PSs into three categories: Fowler's catalog, Model refactorings, and Language-specific refactorings. Refactorings proposed by Fowler fall into the first category (23 PSs), refactoring scenarios applied in design model fall into the second category (3 PSs), and the third category is assigned to refactorings that were applied by specific programming languages involved in the PS (14 PSs). It is important to note that some studies used refactoring operations that belong to two categories. 43 out of 150 refactoring activities were cataloged by Fowler [7] and serve different purposes: composing methods, organizing data, simplifying conditional expressions and method calls, dealing with generalization, and moving features between objects. The other refactorings are either model refactorings or language-specific associated with model or source code artifacts.

As can be observed in Figure 3, some of the refactoring scenarios were studied more frequently than others. The TOP 3 most studied refactoring types are *Pull Up Method*, *Rename Method*, and *Push Down Method*.

Interestingly, while it is expected that PSs opt for popular refactorings, to guarantee their correctness, recent studies that have been mining refactorings [50, 51, 52, 53] have shown that *Pull Up Method*, and *Push Down Method* are among the least used refactorings in practice. We observe that the behavior can be preserved under less or very restrictive conditions depending on the nature of refactoring types. For example, when a class member (method or field) is moved up or down an inheritance hierarchy, or when it is required after the refactoring to have all references to the same variables and methods defined in the same class as before the refactoring as it seems these refactoring operations are the ones that most likely to introduce behavior changes. *Pull Up Method*, and *Push Down Method* are defined as the intention of moving identical methods or attributes, spread in subclasses, up into a superclass, or vice versa, respectively. These refactorings seem to be attractive for researchers to analyze, saying they can be highly useful when removing duplicate code, extracting reusable components, or implementing design patterns. Yet, since they impact several interconnected classes through hierarchies, it is critical to guarantee the refactoring execution correctness. However, developers are found to be rarely performing these types of refactorings through the IDE, instead, it is most likely that they manually move whatever member across hierarchies, and manually fix any unexpected errors that it may cause.

Summary. A variety of refactoring operations have been used in the literature. These refactorings can be classified into three categories: Fowlers catalog, model refactorings, and language-specific refactorings. When testing the proposed behavior preservation approaches by PSs, we observe that some refactoring types such as *Pull Up Method*, *Rename Method*, and *Push Down Method* were studied more frequently than others. The high interest in these refactoring operations in primary studies may indicate their importance in preserving the behavior.

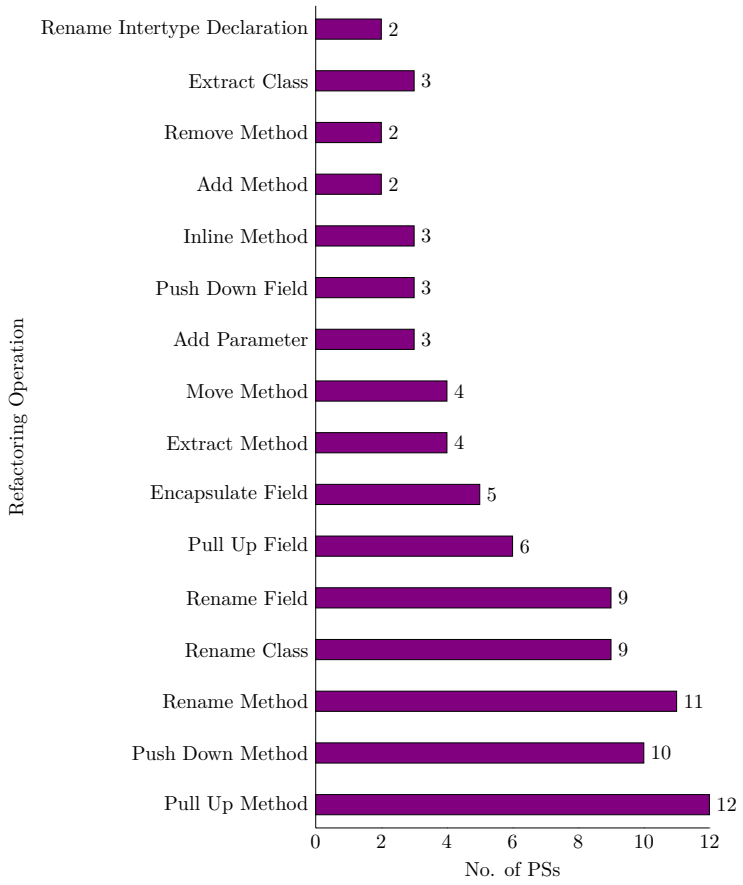


Figure 3: Distribution of Most Used Refactoring Operations.

4.4. RQ3: What approaches were considered by the PSs to test the behavior-preserving transformations in software refactoring?

As discussed in RQ1 and RQ2, refactoring is not restricted to software code, but it also applies to model. Concerning refactoring types used to preserve the

Table 6: Refactorings Identified by Primary Studies and their Classification Schema.

Refactorings	**Classification**		
	Fowler's catalog	**Model refactorings**	**Language-specific refactorings**
Encapsulate Field	✓		
Pull Up Method	✓		
Push Down Method	✓		
Pull Up Field	✓		
Rename Temporary			✓
Move States into Orthogonal Composite State		✓	
Flatten States		✓	
Add Subclass		✓	
Introduce Signature		✓	
Introduce Generalization		✓	
Introduce Subsignature		✓	
Introduce Relation		✓	
Remove Optional Relation		✓	
Remove Scalar Relation		✓	
Split Relation		✓	
Rename Class			✓
Rename Field			✓
Rename Local Variable			✓
Rename Method	✓		
Extract Method	✓		
Extract Class	✓		
Move Class			✓
Change Method Signature			✓
Move Method	✓		
Rename			✓
Move			✓
Introduce USE			✓
Change Function Signature			✓
Introduce Implicit None			✓
Add Empty Subprogram			✓
Safe Delete			✓
Copy Up Method			✓
Extract Local Variable			✓
Add Local Variable			✓
Introduce Block			✓
Insert Assignment			✓
Move Expression			✓
Extract Function			✓
Add Empty Function			✓
Populate Function			✓
Replace Expression			✓
Push Down Field	✓		
Rename Type			✓
Replace Code with Method Call			✓
Move Operation to Listener			✓
Remove Unused Variable			✓
Change Instance Access to Static			✓
Remove Immutable Object Copy			✓
Replace Direct Access with Getter			✓
Replace Instance with isInstance			✓
Remove Parameter	✓		
Replace Field with Method	✓		
Decrease Method Visibility			✓
Replace Direct Access with Setter			✓
Inline Temp	✓		
Consolidate Duplicate Code Fragment	✓		
Rename Constant			✓
Rename Local Variable			✓
Replace Generic Cast with classCast			✓
Replace Generic Cast with isInstance			✓
Replace Method with Method Object	✓		
Change Statement Order			✓
Swap Access Method			✓
Remove Duplicate Assignment			✓
Consolidate Conditional Expression	✓		
Introduce Explaining Variable	✓		
Remove Assignment to Parameters	✓		
Increase Method Visibility			✓
Replace if with Switch			✓
Replace Equivalent Method Call			✓
Introduce Null Object	✓		
Replace Magic Number with Constant	✓		
Wrap (Change) Expression			✓

Table 7: Refactorings Identified by Primary Studies and their Classification Schema (Cont'd).

Refactorings	**Classification**		
	Fowler's catalog	**Model refactorings**	**Language-specific refactorings**
Extract to Function			✓
Extract to Variable			✓
Outer Variable			✓
Variable to Function Parameter			✓
Rename Function			✓
Add Method			✓
Remove Method			✓
Change Method Body			✓
Change Method Modifier			✓
Add Field			✓
Remove Field			✓
Change Field Modifier			✓
Change Field Initializer			✓
Change Static Field Initializer			✓
Rename Intertype Declaration			✓
Inline Method	✓		
Extract Exception Handler			✓
Infer Generic Type			✓
Replace Deprecated Code			✓
Extract Interface	✓		
Extract Subclass	✓		
Generalize Type			✓
Add Variable			✓
Create Accessors for a Variable			✓
Change all Variable refs to Accessors Calls			✓
Remove Class			✓
Move Method across Object Boundry			✓
Extract Code as Method			✓
Change Abstract Class to Interface			✓
Extract Feature into Aspect			✓
Extract Fragment into Advice			✓
Extract Inner Class to Standalone			✓
Inline Class within Aspect			✓
Inline Interface within Aspect			✓
Move Field from Class to Inter-type			✓
Move Method from Class to Inter-type			✓
Replace Implements with Declare Parents			✓
Split Abstract Class into Aspect and Interface			✓
Extend Marker Interface with Signature			✓
Generalize Target Type with Marker Interface			✓
Introduce Aspect Protection			✓
Replace Inter-type Field with Aspect Map			✓
Inter-type Method with AspectMethod			✓
Tidy Up Internal Aspect Structure			✓
Extract Superaspect			✓
Pull Up Advice			✓
Pull Up Declare Parents			✓
Pull Up Inter-type Declaration			✓
Pull Up Marker Interface			✓
Pull Up Pointcut			✓
Push Down Advice			✓
Push Down Declare Parents			✓
Down Inter-type Declaration			✓
Push Down Marker Interface			✓
Push Down Pointcut			✓
Conditional with Polymorphism	✓		
Rename Package	✓		
Move Type	✓		
Extract Superclass	✓		
Add Parameter	✓		
Extract & Move Method	✓		
Extract & Pull Up Method	✓		
Move & Rename Method	✓		
Move Member Type To Toplevel	✓		
Move Member	✓		
Move Inner To Toplevel	✓		
Convert Anonymous To Nested	✓		
Move Instance Method	✓		
Extract Constant	✓		
Extract Temp	✓		
Inline Constant	✓		
Introduce Factory	✓		
Introduce Indirection	✓		
Introduce Parameter	✓		
Introduce Parameter Object	✓		
Promote Temp To Field	✓		

behavior, PSs used a variety of refactoring operations. However, as seen from Figure 3, a number of refactoring operations receive considerable attention due to the fact that these are most likely introduce behavioral changes. Considering the types of software artifacts and refactoring operations used in the PSs, we report, in this section, several approaches for testing behavior preservation of refactoring.

All of the accepted literature publications have reported an approach to preserving the behavior. The approaches vary between using formalisms, applying techniques, developing automatic refactoring safety tools, and performing a manual analysis of the source code. We decided to cluster these approaches as follows: (1) refactoring formalisms and techniques, (2) automated analyses, and (3) manual analysis. Formalism and technique is any behavior preservation approach proposed using a technique or specification. It is not necessarily to be incorporated with a refactoring engine. Automated analysis is any behavior preservation approach that is proposed by incorporating it with a refactoring engine to automate the process. Additionally, we classify the reported approaches into fourteen subcategories. The designed schema for classifying these approaches is depicted in Figure 4. We consider the three already-mentioned classifications as a starting point of the schema, and then classify the reported approaches to each of these classifications. Each PS can belong to one or more subcategories. A detailed overview of these classifications is shown in Table 8. Figures 5 and 6 depict refactoring operations that are overlapped between multiple strategies. Due to the space constraints, we only show the popular refactoring operations used in the literature, namely, *Pull up*, *Push down*, *Extract*, *Move*, *Rename*, *Inline*, and *Encapsulate Field*. As can be seen, these popular refactorings are evaluated using multiple strategies. More details about the overlapped refactorings can be found in our extension package [2]. Detailed descriptions of the approaches are described below.

4.4.1. Refactoring Formalisms and Techniques

This section demonstrates an example that shows some aspects of the Formalisms and Techniques behavior preservation approach. Consider the class `Employee` and its subclass `Salesman`. Class `Employee` declares the `getName`, `getSalary`, `yearlySalary` methods, and Class `Salesman` declares methods `setSSN`, `getSSN`, `getFullName`, `getSalary`, `getSomething`, `toString`, `yearlySalary`, `yearlySalaryIncrease`, `displayYearlySalaryIncrease`, `test1`, and `test2`. Suppose we use generalization-related refactorings (i.e., *Pull Up Attribute* and *Pull Up Method*) to demonstrate this approach. In that case, we notice that one of the strategies listed in Table 8 (i.e., preconditions) needs to be checked before performing refactoring, as follows:

- Methods `setSSN`, `getSSN`, and field `ssn` can be pulled up from class `Salesman` into class `Employee` without affecting program behavior.

[2] https://smilevo.github.io/self-affirmed-refactoring/

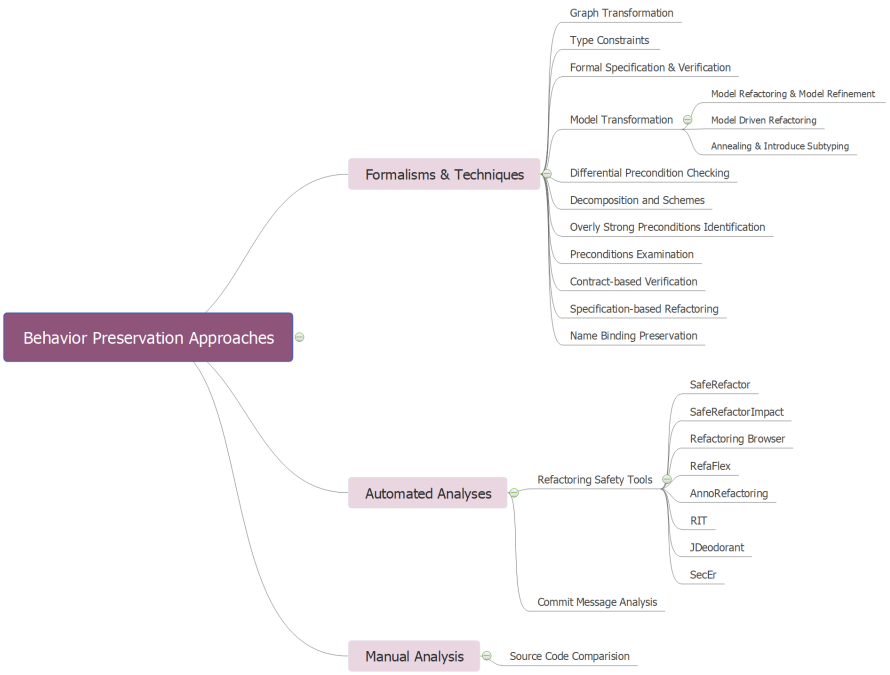
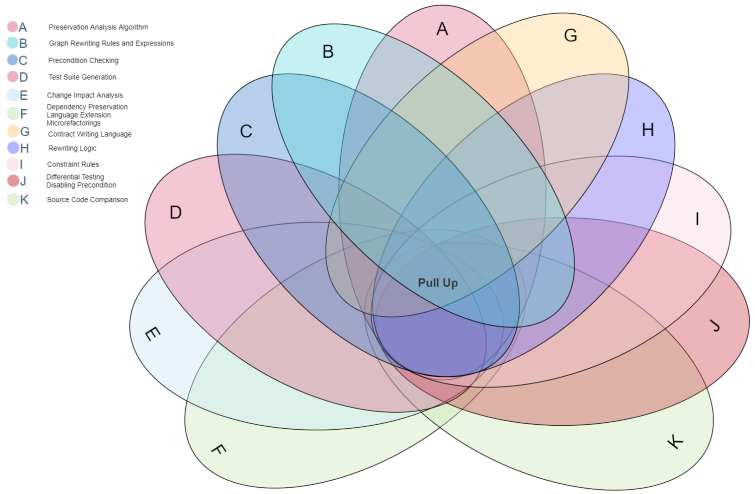


Figure 4: Behavior Preservation Approaches.

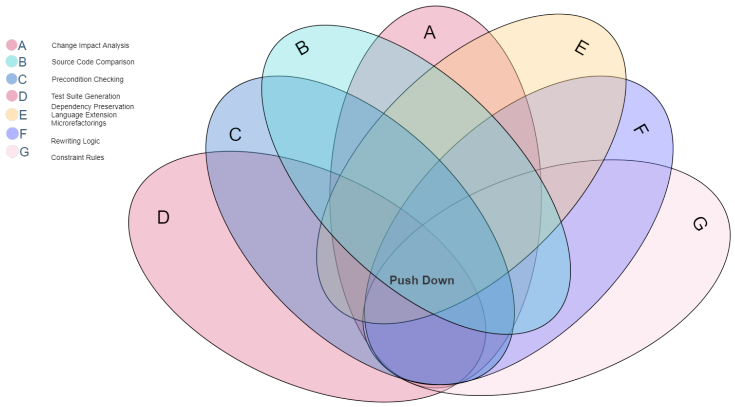
- Method `yearlySalary` cannot be pulled up into class `Employee` because class `Employee` has a method with the same signature defined.
- If method `toString` is pulled up into superclass, there is no compilation error introduced but the program is behaviorally changed. This is because the call `s.toString()` dispatches to a different implementation of the method `toString()`.
- Method `displayYearlySalaryIncrease` cannot be pulled up without pulling up `yearlySalaryIncrease()` because `yearlySalaryIncrease()` is not declared in class `Employee`.

Some aspects of refactoring formalisms and techniques include displaying the violation of refactoring preconditions. For instance, refactoring tools that display violations should: (1) not take longer than a manual refactoring, (2) indicate all locations of precondition violation, (3) show violated preconditions at once, and (4) display the violation relationally.

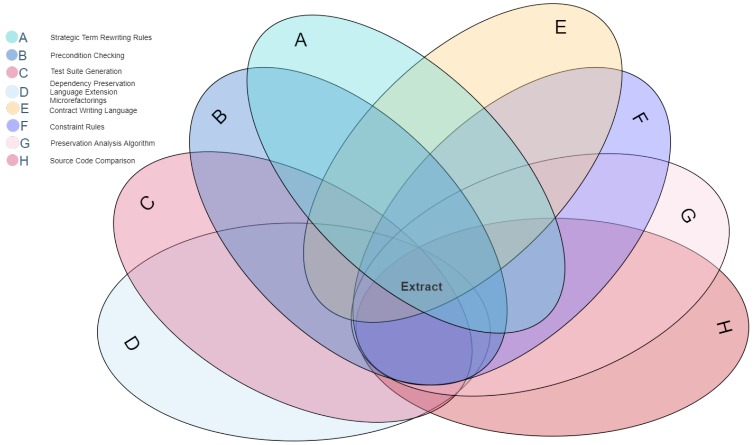
4.4.1.1. Graph Transformation. Existing refactoring tools lack solid specification of the refactoring procedures. Current specifications are defined by examples or by including assertions (pre/postconditions) that are mostly language-specific. To increase the reliability of these tools, a formal model is required



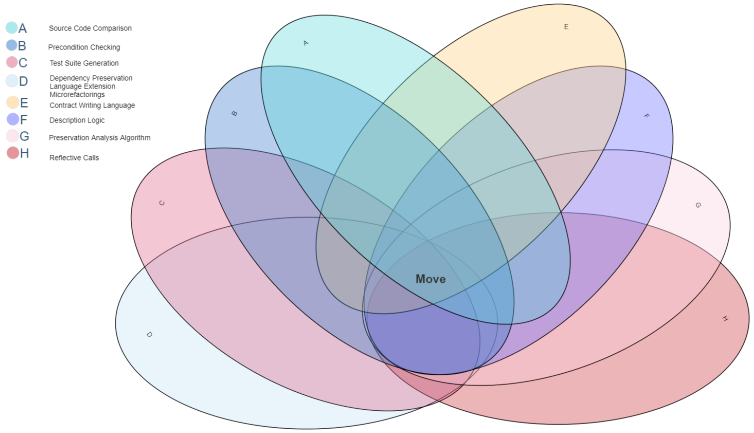
(a) Pull up-related Operations



(b) Push Down-related Operations

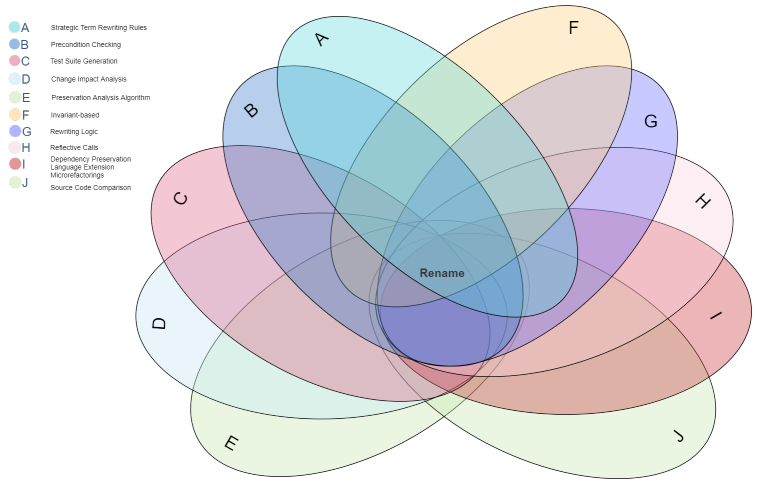


(c) Extract-related Operations

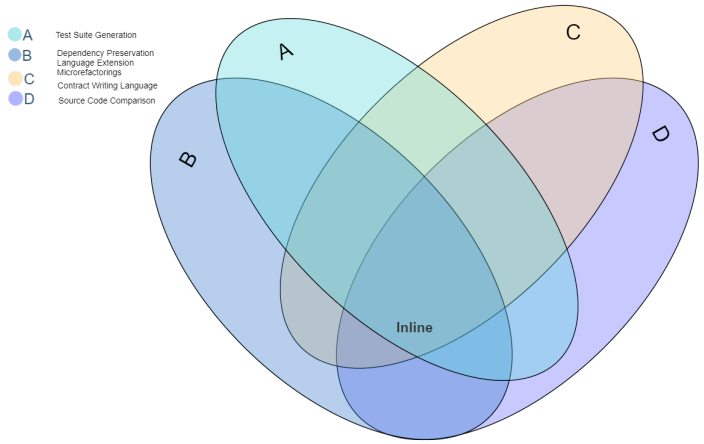


(d) Move-related Operations

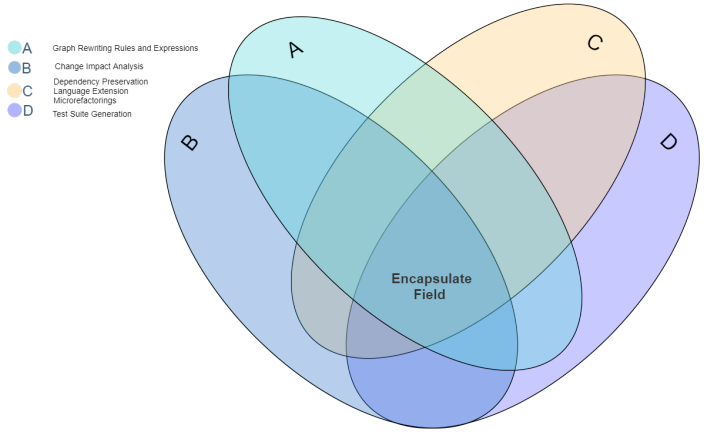
Figure 5: Behavior Preservation Strategies and the Evaluated Refactoring Operations.



(a) Rename-related Operations



(b) Inline-related Operations



(c) Encapsulate Field-related Operations

Figure 6: Behavior Preservation Strategies and the Evaluated Refactoring Operations (Cont).

Table 8: Behavior Preservation Approaches and its Strategies in Related Work.

Study	Year	Approach	Strategy
Roberts et al. [23]	1997	Refactoring Safety Tool	Precondition Checking
Mens et al. [24]	2003	Graph Transformation	Graph Rewriting Rules & Expressions
Tip et al. [22] [36]	2003,2011	Type Constraints	Constraint Rules
Garrido and Meseguer [25]	2006	Formal Specification & Verification	Rewriting Logic
Straeten et al. [26]	2007	Model Transformation	Description Logic
Massoni et al. [27]	2008	Model Transformation	Laws of Programming
Soares et al. [28] [31] [34][37]	2009,2010,2011	Refactoring Safety Tool	Test Suite Generation
Ubayashi et al. [29]	2008	Contract-based Verification	Contract Writing Language
Schäfer et al. [30]	2008	Naming Binding Preservation	Invariant-based
Tsantalis and Chatzigeorgiou [32]	2009	Precondition Examination	Precondition Checking
Schäfer and Moor [33]	2010	Specification-based Refactoring	Dependency Preservation Language Extension Microrefactorings
Tsantalis and Chatzigeorgiou [35]	2010	Refactoring Safety Tool	Precondition Checking
Overbey and Johnson [38]	2011	Differential Precondition Checking	Preservation Analysis Algorithm
Soares et al. [39], Mongiovi et al. [47]	2011,2017	Overly Strong Preconditions Identification	Differential Testing Disabling Preconditions
Jonge and Visser [41]	2012	Name Binding Preservation	Invariant-based
Noguera et al. [42]	2012	Refactoring Safety Tool	Annotation-aware
Thies and Bodden [43]	2012	Refactoring Safety Tool	Reflective Calls
Soares et al. [40]	2013	Refactoring Safety Tool Commit Message Analysis Manual Analysis	Test Suite Generation Keywords-based Search Source Code Comparison
Soares et al. [28], Mongiovi et al. [44]	2009,2014	Refactoring Safety Tool	Change Impact Analysis
Najaf et al. [45]	2016	Annealing & Introduce Subtyping	UML-B Refactoring Rules
Horpácsi et al. [46]	2017	Decomposition & Schemes	Strategic Term Rewriting Rules
Chen et al. [48]	2018	Refactoring Safety Tool	Test Suite Generation
Insa et al. [49]	2018	Refactoring Safety Tool	Test Suite Generation

to support software refactoring which can be expressed by a graph transformation. These formal models should: be a language-independent representation of the source code, preserve certain program properties and include formal analysis of the assertion to ensure the completeness of the specification. Further, since refactoring tools represent the source code by abstract syntax tree and any refactoring activity is supposed to change that graph, there is a need to have a formal specification for refactoring that corresponds with a number of graph rewriting rules [24].

In a study survey, Mens and Tourwe [9] summarize these formal properties by showing the correspondence between refactoring and graph transformation as shown in Table 9.

Table 9: Formal Properties of Graph Transformation (Extracted from Mens and Tourwe [9]).

Refactoring	**Graph Transformation**
software artifact	graph
refactoring	graph production
composite refactoring	composition of graph productions
refactoring application	graph transformation
refactoring precondition	application precondition
refactoring postcondition	application postcondition

4.4.1.2. Type Constraints. Tip et al. [22] [36] propose using type constraints that depend on interprocedural relationships between variable types. A con-

straint variable can be one of the following: (1) type of constant, (2) type of expression, (3) type of declared method, and (4) type of declared field. To ensure the preservation of program behavior, each refactoring should be associated with a set of preconditions that must be satisfied. Type constraints mechanism verifies the preconditions and determines the source code that is allowed to be modified.

4.4.1.3. Formal Specification and Verification. Due to the lack of accurate specification of the preconditions and lack of proof of the correctness of the refactoring tools, Garrido and Meseguer [25] introduce an equational (rewrite logic) semantics-based approach that fulfills two essential goals: (1) formally specifying Java refactorings, and (2) proving behavior-preserving of refactorings with respect to the language's formal semantics. The implementation of this approach is based on the rewrite logic executable semantics of Java refactorings in Maude language. They show the Maude specification of push down method, pull up field, and rename temporary Java refactorings along with providing a mathematical proof of the correctness for two of those refactoring operations. They particularly prove that these refactorings preserve the program behavior with reference to the formal Java semantics.

Consider the formal specification of *Pull Up Attribute* defined in [25]. By applying this refactoring operation on field `ssn` to move the field to the class `Employee`, the following preconditions must hold in order for transformation to be carried out successfully.

- There is a class named `Employee`.
- Class `Employee` has at least one subclass.
- Class `Employee` does not define the field `ssn`.
- Subclass of `Employee` defines the field `ssn`.

These preconditions are checked by `preconditionsPullUpFieldHold` operation and applied by operation `applyPullUpField` in the formal specification listed in [25].

4.4.1.4. Model Transformation.

Model Refactoring and Model Refinement. Straeten et al. [26] differentiate between model refactoring and refinement as follows: model refactoring aims at improving the model structure while preserving its behavior, while model refinement aims at providing more detail to an existing model. The main purpose of this formalism is to investigate the relation between the behavior inheritance consistency and behavior preserving properties of a refined model and a refactored model respectively. These model transformation activities are used to manipulate models, and are supported by a practical formalism that detects the behavior consistency between a refined model and a refactored model. This is achieved by the developed plug-in in a UML CASE tool. These

two kinds of model transformation need to be complemented by model inconsistency management to avoid any possibility of inconsistency between models after the transformation activity. Straeten et al. [26] used reasoning capabilities of descriptive logics (DLs) [54] to detect behavioral inconsistencies during model refinement and behavioral preservation violation during model refactoring. In other words, they check the behavior inheritance consistency between superclass and its subclass and then ensure that this behavior consistency is preserved between refactored classes in an inheritance hierarchy.

Model-Driven Refactoring. Some of the popular model-driven development approaches (e.g. Round-trip Engineering) generate changes to programs from a model, which requires manual updates, making the evolution costly [27]. To avoid any manual update activity on the source code, Massoni et al. [27] propose a formal approach to refactor programs in a model-driven manner in a semi-automatic way which guarantees behavior preservation of the target program. The precondition for applying this approach is to guarantee that the source code is in conformity with the object model. In this approach, Alloy is used as the model transformation system, where each primitive Alloy model transformation from the catalog is associated with a strategy to refactor the program. This results in a program consistent with the refactored program (i.e. behavior preservation). This formal approach is guided by laws of programming that have been proven to be behavior-preserving.

Annealing and Introduce Subtyping. Najafi et al. [45] proposed annealing and introduce subtyping rules as refactoring rules which can improve the design from an abstract specification written in UML-B. These rules are similar to refactoring rules proposed by Fowler et al. [7]. Annealing adds structure to a specification by splitting a class into two classes (aiming for more fine-grained classes); whereas introducing subtyping establishes a relationship between classes with many features in common (with the aim of increasing reusability). The rules are behaviorally preserved as it ensures that any software design produced will be correct with respect to the original specification.

4.4.1.5. Differential Precondition Checking. Due to the importance of setting preconditions to help guarantee behavior-preserving program transformations for automated refactorings, Overbey and Johnson [38] propose a technique called Differential Precondition Checking. This technique has the added advantages of being language independent and reusable in a library. It checks for preservation by first analyzing the source code and creating program representation. Before validating user input, it constructs a program graph as a semantic model (i.e. initial model). It then produces a new program representation for the modified source code for the purpose of checking compilability of the refactored program. For this new program representation, it also constructs a derivative semantic model. The following step is to perform preservation analysis by comparing the two semantic models. If the differential precondition checker determines that

the transformation is behavior preserving, the modification will be applied to the source code. Otherwise, the refactoring will not be considered.

By way of illustration, Overbey and Johnson [38] show the differences between the traditional precondition checking and the differential checking for *Pull Up Method* refactoring. For the traditional version, the method needs to be moved from subclass to its superclass, replacing all occurrences of superclass with `this`. Using preservation rule for the differential version, however, this refactoring is composed of two smaller refactoring operations: (1) *Copy Up Method* to move a method to its superclass and replace all occurrences of the superclass with `this` and (2) *Delete Overriding Duplicate* to delete the original method from the subclass using the preservation rule in [38].

4.4.1.6. Decomposition and Schemes. Some of the previously built complex refactoring tools were solidly developed, but not totally accurate in guaranteeing the correctness of the transformation these tools implement, which have resulted in introducing bugs to the system. In order to solve this problem, Horpácsi et al. [46] propose to decompose complex refactoring transformation into a series of prime refactorings that can also be expressed as instances of refactoring schemes, and then verified based on formal program semantics. This way, the transformations become simple and more easily verified.

4.4.1.7. Overly Strong Precondition Identification. Soares et al. [39] propose an approach to identify overly strong conditions in refactoring implementations as these conditions may prevent behavior-preserving transformations. This formal specification helps in guaranteeing program preservation. The process of checking overly strong conditions begins with an automatic generation of Java programs as test inputs using a program generator called JDolly. For each program generated by JDolly, the same refactoring is applied by using three different refactoring implementations (i.e., Eclipse, JRRT, NetBeans). Then, the outputs of the refactoring implementations are compared. To evaluate whether a transformation is behavior-preserving, SafeRefactor is used to identify behavioral changes in transformation. If one implementation rejects the transformation, and the other implementation accepts it with the conformation from SafeRefactor tool that it is behaviorally preserved, this is an indication that the first implementation that rejects the transformation contains an overly strong condition. This technique is also called Differential Testing (DT) [55].

For an example of such an overly strong condition, suppose we apply *Rename Method* refactoring to rename method `getFullName` to `getName`. If we apply this refactoring using Eclipse, we get the following warning message: *Problem in 'Salesman.java'. The reference to `getName` will be shadowed by a renamed declaration.* After applying the transformation, the `test2` method outputs `John Smith` instead of `John`. This transformation exposes a behavioral change after ignoring a warning message. Similarly, NetBeans applies the transformation.

By applying this refactoring using JRRT, however, the transformation preserves behavior. JRRT adds a `super` access to method `getName` inside `test2` to ensure that the resulting program correctly refactors the source program.

We notice that Eclipse rejects the transformation, and NetBeans and JRRT apply it with the conformance from SafeRefactor tool that it is behaviorally preserved. Thus, by comparing the results of Eclipse, JRRT, and NetBeans, it indicates that Eclipse has an overly strong condition because it rejects useful behavior preserving transformation.

In the following study that complements this work, Mongiovi et al. [47] propose a new technique called Disabling Preconditions (DP) to detect overly strong preconditions. The process starts with using JDolly as test inputs (Step 1). For each generated program, the refactoring engine is used to apply the transformations. In Step 2, authors collected the messages reported by the refactoring engine about the rejection of certain refactoring transformations. The next step is to manually inspect the code fragments and its related precondition for the purpose of disabling the execution of the precondition (i.e., DP technique). Step 5 involves reapplying the same transformation with a disabled precondition. After ensuring that the refactoring implementation applies the transformation and this transformation is behaviorally preserved according to SafeRefactorImpact, DP technique classifies a precondition as overly strong precondition.

4.4.1.8. Behavior Preservation Preconditions Examination. Tsantalis and Chatzigeorgiou [32] propose a methodology to preserve the behavior of the code by examining a set of preconditions when applying *Move Method* refactoring. These preconditions should be satisfied in order to avoid behavioral changes. Tsantalis and Chatzigeorgiou [32] formally define a set of auxiliary functions that describe behavior preservation preconditions as follows:

- A class should not inherit a method having a matching signature with the moved method. This action will lead the inherited method to override causing behavioral changes of the target class and its derived one. The moved method needs to be renamed to resolve the issue.
- When moving a method, the method should not override an inherited method. The original method should be kept as delegate to the moved method.
- When moving a method, the method should have a valid reference to its target class. The moved method can have a reference via its parameters or fields in the original class.
- When moving a method, the method should not be synchronized. Moving the synchronized method might cause concurrency issues to the original class's objects.

4.4.1.9. Contract-based Verification. Ubayashi et al. [29] proposed the notion of Refactoring by Contract (RbC) to verify the applied refactoring based on contracts. The contracts in RbC consist of preconditions (i.e., which conditions can be applied), postconditions (i.e., which conditions should be verified after refactoring), and invariants (i.e., what conditions refactoring should preserve).

This contract is described in Contract Writing Language (COW), to describe a predicate based on first-order logic. Another study by Dao et al. [56] verified the execution preservation of refactored program which is performed by design patterns. The authors proposed consistent rules (i.e., pre/postconditions) to verify if the execution of the original program and refactored one is preserved the same constraints in the evolution process.

4.4.1.10. Specification-based Refactoring. Because the precondition-based approach is hard to maintain, Schäfer and Moor [33] presented an approach that is based on the concepts of dependency preservation, language extensions, and microrefactorings. The authors pointed out that the approach is powerful enough to provide high-level and precise specifications of many of the refactorings. The author validated their implementation on Eclipses own extensive test suite.

4.4.1.11. Name Binding Preservation. Since name binding associates identifiers with program code elements, it forms a semantic concern that should be preserved by refactorings. Schäfer et al. [30] pointed out the two limitations in current refactoring tools: (1) too weak preconditions that lead to unsoundness where names do not bind to the correct declarations after renaming, and (2) too strong preconditions that prevent renaming of certain programs. The authors proposed an invariant-based approach for name binding. In another study, Jonge and Visser [41] focused on the behavior preservation of static name bindings by implementing a name binding preservation criterion that reuses the name analysis defined in the compiler front-end. This way, even when the language evolves, the semantics assumed by the refactoring tool is guaranteed to be consistent with the semantics implemented in the compiler.

4.4.2. Automated Analyses

4.4.2.1. Refactoring Safety Tools.

SafeRefactor. With the emerging use of refactoring tools, evidence show that these tools do not always preserve behavior since they may lead to erroneous transformations [34] [28]. In order to avoid such refactoring errors, Soares et al. [28] developed a tool named SafeRefactor to check refactoring safety. It generally works by identifying behavioral changes in transformations of sequential Java programs and then generating a test suite for capturing unexpected behavioral changes. This process splits into five major sequential steps. After receiving two versions of the program as an input, a static analysis detects common methods in both the source and target programs (step 1). The next step involves generating unit tests for methods identified in step 1 to pinpoint the incorrectly performed refactorings. In step 3, the tool executes the generated test suite on the source program and then runs the same test suite on the target program (step 4). The last step validates whether the transformation introduces behavioral changes: If a test runs successfully in one program and fails in the other, the tool identifies a behavioral change. Otherwise, no behavioral changes will be detected and the transformation is behaviorally preserved.

SafeRefactorImpact. The SafeRefactor tool has been extended, and includes AspectJ support [37], uses change impact analyzer called SAFIRA, and generates a test suite only for the methods impacted by the transformation [57, 44]. SafeRefactor was renamed SafeRefactorImpact in [44]. This tool works by: (1) comparing the original and modified programs to identify entities (methods) impacted by the change, (2) performing a change impact analysis technique for the impacted methods in both program versions identifying methods that can be behaviourally changed after the transformation, (3) generating a test suite for the common methods identified in the previous step, (4) executing the test suite before and after the transformation, and (5) evaluating the results of the transformation to determine whether the transformation is behavior preserving.

Mongiovi et al. compare these tools in [44] with respect to several criteria: program correctness, performance, number of methods considered for test generation, change coverage, and relevant tests generated. Their findings show that the extended tool generates better results.

Refactoring Browser. Roberts et al. [23] developed a tool called Refactoring Browser, which uses a set of preconditions to ensure a safe and a correct refactoring implementation. This tool was designed solely to automate refactorings for the Smalltalk language. The tool is used regression testing to assure that refactorings indeed do not alter the programs behavior. In order to preserve the behavior of the program, each refactoring is associated with a reused set of preconditions that must be checked by the compilation framework in VisualWorks. For instance, to successfully implement *Add Method* refactoring, the method name should not conflict with a method defined in the class.

RefaFlex. Since reflective calls are the threats to the validity of refactorings, Thies and Bodden [43] proposed RefaFlex Eclipse plugin tool for reflective Java programs to ensure refactoring safety. The tool used dynamic analysis to log reflective calls during test runs and then utilized the information to prevent the execution of refactorings that could alter the program's behavior.

AnnoRefactoring. During the condition checking phase of the performed refactoring, annotations can break the behavior preservation as the annotation's restriction can be ignored which no longer guarantee the preservation of the domain-specific mappings. To address this problem, Noguera et al. [42] developed an annotation-aware refactoring tool that is integrated with Eclipse to document the domain dependencies that the annotations introduce. Instead of augmenting the refactoring preconditions with the annotation behavior specification, the authors implemented the annotation behavior preservation as post-conditions. Since the refactoring-aware annotation is considered as dependency preservation problem, checking whether the dependencies were maintained after refactoring is crucial.

RIT. Chen et al. [48] proposed an Eclipse plugin tool named Refactoring Investigation and Testing (RIT) in order to validate refactoring changes and ensure that the changes behave as intended. For each set of identified refactoring changes, The tool analyzed the original and edited version of the programs, and then detect tests whose behavior might have been modified by refactoring edits. The developed tool helps developers detect refactoring edits responsible for test failures.

JDeodorant. Tsantalis and Chatzigeorgiou [35] proposed a technique, implemented as an Eclipse plugin, that extracts refactoring suggestions introducing polymorphism to ensure the behavior preservation based on the examination of a set of preconditions. This technique helps with eliminating the state-checking problem that impacts code quality, and its maintenance requires significant effort.

SecEr. Insa et al. [49] developed a Sofware Evolution Control for Erlang (SecEr) tool to automatically obtain a test suite that specifically focused on comparing the old and new versions of the code to check the behaviour preservation. Differently from SAFIRA [57, 44] that focused on refactoring as a cause of the change, SecEr is independent of the cause of the changes, being able to analyze the effects of any change in the code regardless of its structure. All the analyses performed by the tool are transparent to the user except that it requires user intervention when identifying point of interests in both the old and the new versions of the program.

4.4.2.2. Commit Message Analysis. One of the approaches to analyze refactoring activity on software repositories is by analyzing commit messages. Ratzinger [58] and Ratzinger et al. [59] propose this simple and fast approach to detect refactoring activity between a pair of program versions to determine whether a transformation is behavior preserving. They identified refactorings based on a set of keywords existing in the commit message. In particular, they focus on the following terms in their search approach: *refactor, restruct, clean, not used, unused, reformat, import, remove, replace, split, reorg, rename, and move.*

Few commit messages containing some of these terms are extracted from the Hadoop[3] project, as illustrated in the following comments:

"*1. HADOOP-9805. Refactor RawLocalFileSystem rename for improved testability. Contributed by Jean-Pierre Matsumoto.*"

"*2. HDFS-7743. Code cleanup of BlockInfo and rename BlockInfo to BlockInfoContiguous. Contributed by Jing Zhao.*"

[3]https://github.com/apache/hadoop

4.4.3. Manual Analysis

Murphy-Hill et al. [3] identifies refactoring activities by manually analyzing and comparing the source code before and after the commit. For example, to check whether each file before and after the commit preserved behavior, evaluators first review the code to understand the syntax and semantic changes and then use diff tool to help them analyze the transformation. After that, they classify the code changes as either refactoring (such as *Move class* or *Inline method*) or non-refactoring (such as *Add null check*). In case of disagreements on whether the applied refactoring changed the behavior, evaluators discussed them until agreement was reached.

Soares et al. [40] compared and evaluated three approaches, namely, manual analysis, commit message, and dynamic analysis (SafeRefactor approach) to analyze refactorings on open source repositories, in terms of behavioral preservation. They found, in their experiment, that manual analysis shows the best results in the comparison and is considered the most reliable approach in detecting behavior-preserving transformations.

Summary. Many behavior preservation approaches have been proposed in the literature. The approaches vary between using formalisms and techniques, developing automatic refactoring safety tools, and performing a manual analysis of the source code. Researchers are biased toward using precondition-based and testing-based approaches although there are other techniques (e.g., graph-based) that have some potential and perhaps it is effective for certain problems that have not yet well-explored. Several possible strategies can be combined to better detect any violation of the program semantics. Formalism and technique approaches are mainly precondition-based, graph-based, model-based, and decomposition-based techniques; automated approaches either rely on testing, preconditions, or keywords. Manual approach is comparison-based in which the source code has been compared before and after the commit.

4.5. RQ4: What evaluation methods were used in the PSs to assess the proposed behavior preservation approaches?

Table 10: Evaluation Methods Used by the Primary Studies.

Methods	No. of PSs	PSs
Comparison-based	5	[38] [40] [44] [47] [49]
Empirical-based	13	[22] [28] [31] [34] [39] [37] [43] [36] [33] [30] [41] [48] [42]
Formal Specification-based	7	[27] [26] [46] [25] [29] [45] [30]
Qualitative-based	1	[32]
Independent assessment-based	2	[32] [35]

Except for [24] and [23], all of the PSs used certain evaluation methods to

validate their approach. We identified five different evaluation method categories. The applied methods include comparing the approach against others [44][47][38][40] [49], running an experiment in one or more refactoring transformations [28][31] [34] [39][22] [37] [43] [36] [33] [30] [41] [42], presenting a formal specification for correctness of refactorings [27][46] [26] [25] [29] [45] [30], using qualitative analysis [32] [48], and independent assessment [32] [35]. The authors of [24] don't evaluate their approach, but they plan to validate their approach in the future by the following steps: (1) converting code into a graph, (2) applying graph transformation approach to the graph, and (3) verifying the preconditions for two refactoring operations. Table 10 shows the distribution of the PSs over the evaluation methods and the descriptions are detailed below.

4.5.1. Comparison-based evaluation

Regarding the first evaluation method, the authors of the PSs compare their approach to other existing methods. Overbey and Johnson [38] evaluate their approach in three refactoring tools from two different perspectives: the expressivity of the preservation specifications and the performance of differential precondition checking approach compared to a traditional one. Mongiovi et al. [44] compare SafeRefactorImpact with SafeRefactor in terms of the similarity of the detected behavioral changes, total time to evaluate the transformation, number of impacted methods, and the change coverage of the generated test suites. Soares et al. [40] compare the three approaches (i.e., SafeRefactor, commit messages analysis, and manual analysis) in terms of identifying all behavior preservation, correctness of the identified behavior preservation, and accuracy of the obtained results. Mongiovi et al. [47] evaluate the approach by comparing bugs detected by Disabling Preconditions (DP) and Differential Testing (DT) techniques. Insa et al. [49] compared SecEr with the already available debugging and testing techniques used when behaviour preservation is checked in an Erlang project.

4.5.2. Empirical-based evaluation

For empirical-based evaluation, Soares et al.[34] ran the experiment in 24 refactoring transformations using real Java applications and transformations applied by refactoring tools. Soares et al. [31] also experimented 16 refactoring cases which successfully detected more than 93% of errors presented by traditional refactoring tools. Soares et al. [28] evaluate their approach against 9 transformations and the approach did not produce any errors compared to 5 wrongly applied transformations by best refactoring tools. Soares et al. [39] assessed their approach by performing an experiment in 27 refactoring operations of three refactoring tools: Eclipse, JRRT, and NetBeans. Tip et al. [22] [36] implemented only *Extract Interface* refactoring in Eclipse to test the proposed approach. Soares et al. [37] evaluated the proposed technique in 8 refactorings applied by Eclipse, 23 design patterns, 2 case studies, and 2 JML compilers. Schäfer et al. [30] [33] evaluated the correctness of their refactoring engine in Eclipse test suite. Chen et al. [48] applied RIT in 3 Java open source projects that have regression test suites. Jonge and Visser [41] assessed their approach

by implementing refactoring for 3 different languages, namely, Mobl, Stratego, and subset of Java. For Mobl and Stratego, they used the existing compilers, whereas for Java subset, they implemented the compiler from scratch. Noguera et al. [42] used a prototype extension of the Eclipse IDE's to demonstrate their approach using three annotation libraries: JPA, Aspect5J, and Simple XML. RefaFlex was evaluated in [43] with 21,524 refactoring runs on 3 open source programs. Their approach prevented 1,358 non behavior preservation transformations.

4.5.3. Formal specification-based evaluation

In four PSs, including [27] [46] [26] [25], the approaches were evaluated by formally specifying and verifying the refactoring to ensure that these refactorings are behaviorally preserved. Ubayashi et al. [29] evaluated their approach by writing contracts using first-order predicates. Their approach provided good results and most of these contracts can be generated automatically. Najafi et al. [45] evaluated their refactoring rules by applying them to an adapted study of the *Mass Transit Railway System.*

4.5.4. Qualitative-based evaluation

In [32], Tsantalis and Chatzigeorgiou assessed their approach using open-source Java projects in four different ways: (1) performing a qualitative analysis of the refactoring suggestions, (2) using software metrics related to coupling and cohesion, (3) having an independent assessment on the refactoring suggestion, and (4) evaluating the efficiency by measuring the computation time with different size of open-source projects.

4.5.5. Independent assessment-based evaluation

In [32] and [35], the proposed approach was evaluated by an independent designer for the system that he developed. The designer provided feedback on the refactoring result from the proposed approach.

> *Summary.* With regards to the evaluation methods used in the literature to validate the proposed behavior preservation approaches, PSs used comparison-based, empirical, formal specification-based, quality-based, and independent assessment-based evaluation methods. The majority of PSs empirically evaluate their approaches, and only one study opted for quality-based and independent assessment-based approaches.

5. Discussion and Open Issues

To ensure that the transformation is behaviorally preserved, we recommend incorporating refactoring tools with the following dimensions:

- Preconditions & Postconditions & Invariant: These properties are used to flag potential violations, such as incompatible signatures in member function redefinition, type-unsafe assignments, or indistinct class and naming [8]. Refactoring tool support needs to determine the number of the preconditions, postconditions, and invariants for each refactoring operation applied by including efficient algorithms for checking these assertions. Although Opdyke proposed a set of refactoring preconditions, there was no formal proof of the correctness of these conditions. Developers should invest into developing more comprehensive refactoring tools by (1) adding library containing these assertions to check refactoring so that any refactoring engines for different languages can use this library to test refactoring implementation; and (2) adding formal proofs of the correctness of these assertions to raise the confidence that these set of refactoring help in ensuring that that the transformations preserve the behavior. Additionally, the calculation of pre and postcondition scenarios is time-consuming and error-prone if it is done manually. Future researchers are encouraged to adopt tools to automatically calculate these assertions and verify the program evolution process.

- Quality Improvement: In software engineering, maintaining quality is always a top priority. As development progresses and flaws inevitably begin to emerge, they generate what is known as code smells, various indicators that code needs to be refactored or replaced, and can be helpful in identifying problem areas that need to be refactored. Due to the number of design choices, it is challenging to choose the optimal refactorings, maximising the quality of the resulting program while minimizing the cost of behavior preservation transformation. Besides ensuring behavior preservation of the program, it is also advised to check if the resulting program improves the quality of the original program. For instance, the resulting program showcases reusability and provides trustworthiness by reducing the complexity of the program.

- Developer Perception: Research in preserving the behavior in software refactoring thus far focuses on proposing approaches assuming that the developer's main intention is to perform pure refactoring. Several studies [1, 2, 60, 52] have been conducted to better understand the motivation behind refactoring (e.g., improving the internal and external structure of the code , removing code smells, etc). Current approaches have not integrated developers' perception while preserving the behavior of refactoring activities. Researchers should explore developers' insight and experience (e.g., when and how) because they are essential in the behavior preservation process.

- Automated Testing: Some studies [PS7, PS10, PS12, PS13, PS16, PS19, PS23] discussed using testing to ensure behavior preservation but with limited coverage. To increase refactoring safety, it is needed to incorporate a solid test suite to the traditional refactoring steps in order to pinpoint non

behavior-preservation transformations. That involves generating testing for refactoring applied at different levels of granularity, and taking into account the hierarchy or other object-oriented property.

- Tools Availability and Extensibility: As we noticed in relation to studies [PS1, PS7, PS14, PS21, PS22, PS23, PS28], there is a lack of available tools to support the behavior preservation. Researchers will not be able to adopt behavior preservation approaches because these tools are not available. As a result, it will make it hard to extend the proposed approaches (e.g., support more refactoring operations, add additional set of preconditions, etc). Additionally, Eclipse plugin tools require user interaction to select projects as inputs to trigger refactorings, which is impractical for a study requiring a high degree of automation since multiple releases of the same project must be imported to Eclipse to check whether the behavior is preserved or not. Further, while some of the current tools warn developers of non-behavior preservation transformations, these tools could be complemented with a compensation transformation that possibly preserve the behavior. To move the research forward in this area, researchers are advised to implement a full-featured refactoring engine such as integrating the tools with control version systems like Git or Subversion to easily compare code among several versions and to open source these tools and allow people to replicate and extend them.

- Broader Applicability: Today, a wide variety of refactoring tools automates several aspects of refactoring. However, ensuring the behavior preserving property when building tool-assisted refactoring is challenging. It is acknowledged that refactoring tools should support the following five characteristics: automation, reliability, configurability, coverage, and scalability. Integrating behavior-preserving nature reduces the need to perform testing and debugging. As shown in Figure 4, several studies presented many approaches to preserve the behavior. However, we still must understand which approaches are the most effective. While the primary studies proposed refactoring preservation approaches, these approaches should not be language-specific, domain-specific, and refactoring operation-specific. One important research direction is to generalize the behavior preservation approach across multiple languages and multiple domains, and enable semi-automatic formal verification. Researchers are encouraged to explore such interests together with the practice of preserving the behavior in software refactoring.

The above mentioned open issues are listed in Table 11. A summary of the findings is reported in Table 12. We observe that researchers are biased toward certain approaches. As can be seen from the table, researchers extensively used a precondition-based approach. Testing-based is also popular due to the fact that researchers are probably implementing preconditions to test whether the transformation is behaviorally preserved between multiple versions. However, there are other techniques (e.g., graph-based) that have some potential and

perhaps it is stronger or effective for certain problems that have not yet explored. Incorporating these specifications in IDE refactoring engines, developers and researchers can revisit existing refactoring tools and extend them.

Recent refactoring research has been taking developer-centric strategies to understand how developers refactor and document their refactorings in practice [61, 62]. Such research has been driven by the rise of several refactoring mining tools [2, 63, 50]. Mining the history of previous changes unlocked another dimension of how we should perceive refactoring: Instead of *dictating* how refactoring should be performed and preserved, we can reverse engineer how developers refactor their code and verify the correctness of their operations. Such findings require accurate detection of refactorings, which can be assured by recent studies, as they are reaching a significant precision [50]. Furthermore, the list of mined refactorings has revealed the existence of refactoring types that were absent from studies handling the behavior preservation [53].

6. Implication

The main implications of this study are as follows:

6.1. Implication for practitioners:

- **Promoting the adoption of behavior preservation approaches in practice.** Due to the growing complexity of software systems, there has been a dramatic increase and industry demand for tools and techniques on software refactoring. Refactoring studies are used in industrial settings and considered objectives beyond improving design to include other non-functional requirements. Thus, challenges to be addressed by refactoring work nowadays include testing the correctness of applied refactorings. Recent studies (e.g., [64, 3, 65]) show developers under-using automated refactoring tools due to the lack of trust, unawareness, and usability problems. To mitigate this issue, our study reveals several behavior preservation approaches that can be explored to reduce verification effort. For example, developers can use the tool Refactoring Investigation and Testing (RIT) to (1) help them detect refactoring changes responsible for test failures and validating the correctness of the refactored version of the program without the need to rerun the entire regression test suites, and (2) help developers focusing on the long-term management of accidental complexities created by quick design and implementation (e.g., refactoring to reduce technical debt).

- **Identifying the needed information to the refactored code.** The awareness of such behavior preservation approaches assist programmers in distinguishing precondition violations from warning and advisories without wondering if there are any issues with the applied refactoring. Additionally, it gives programmers an indication of the amount of work required to fix the problem, and so the programmers can determine whether the violation means that the code can be refactored with a few minor changes or not.

Table 11: Open Issues on Behavior Preservation Studies.

Issue	PSs	Open Issue
I1 - Assertion Precondition Postcondition Invariant	 PS1, PS2, PS3, PS4, PS8, PS11, PS15, PS17, PS18, PS21, PS26 PS2, PS8, PS17, PS21 PS8, PS9, PS21	- Researchers can add libraries containing these assertions to test refactoring implementation - Researchers can add formal proofs of the correctness of these assertions to raise developers' confidence
I2 - Quality Improvement	PS11	- The studies do not establish an explicit connection between behavior preservation approach and quality, showing there is an opportunity for further studies
I3 - Developer Perception	N/A	- The use of developers' perception and knowledge about refactoring can help to improve refactoring process, tools, among other - It is essential to evaluate the participation of developers in preserving the behavior, using developers' insights and experiences to improve the process
I4 - Automated Testing	PS7, PS10, PS12, PS13, PS16, PS19, PS23	- It is an open theme for researchers to incorporate a solid test suites to test behavior preservation
I5 - Tool Availability & Extensibility	PS1, PS7, PS14, PS21, PS22, PS23, PS28	- There are many opportunities to propose/improve behavior preservation automated tools - We need to integrate the tools with control version systems such as Git or Subversion
I6 - Broader Applicability	N/A	- We need to explore which approaches are most effective in behavior preservation - Production of refactoring-agnostic approach - Implementation of language-independence of refactoring schemes - There are many opportunities to research a low explored refactoring operations, most used refactorings and their relationship with behavior preservation - Several possible strategies can be combined to better detect any violation of the program semantics - Identification of the appropriate and most reliable evaluation methods to validate the future behavior preservation approaches. - Refactoring tools could be complemented with a compensation transformation that possibly preserve the behavior. - Researchers are encouraged to explore the above-mentioned aspects together with the practice of preserving the behavior in software refactoring.

Table 12: Summary of Behavior Preservation Approaches in the Primary Studies.

Study ID	Study	Software Artifact		Language	No. of Ref.	Refactoring Classification			Approach	Evaluation Method
		Code	Model			Fowler's catalog	Model ref.	Language-specific		
PS1	Roberts et al.	Yes	No	Smalltalk	18	Yes	No	Yes	Refactoring Safety Tool	Not Mentioned
PS2	Mens et al.	Yes	Yes	Not mentioned	2	Yes	No	No	Graph Transformation	Future Validation Tool
PS3	Tip et al.	Yes	No	Java	7	Yes	No	Yes	Type Constraint	Empirical-based
PS4	Garrido & Meseguer	Yes	No	Java	3	Yes	No	Yes	Formal Specification & Verification	Formal specification-based
PS5	Straeten et al.	No	Yes	UML	3	No	Yes	No	Model Transformation	Formal specification-based
PS6	Massoni et al.	No	Yes	Alloy	7	No	Yes	No	Model Transformation	Formal specification-based
PS7	Soares et al.	Yes	No	Java	1	Yes	No	No	Refactoring Safety Tool	Empirical-based
PS8	Ubayashi et al.	Yes	No	AspectJ	27	No	No	Yes	Contract-based Verification	Formal specification-based
PS9	Schäfer et al.	Yes	No	Java	3	Yes	No	No	Naming Binding Preservation	Formal specification-based
PS10	Soares et al.	Yes	No	Java	8	Yes	No	Yes	Refactoring Safety Tool	Empirical-based
PS11	Tsantalis & Chatzigeorgiou	Yes	No	Java	1	Yes	No	No	Precondition Examination	Quality-based Independent assessment-based
PS12	Schäfer & Moor	Yes	No	Java	3	Yes	No	No	Specification-based	Formal specification-based
PS13	Soares et al.	Yes	No	Java	12	Yes	No	Yes	Refactoring Safety Tool	Empirical-based
PS14	Tsantalis & Chatzigeorgiou	Yes	No	Java	1	Yes	No	No	Refactoring Safety Tool	Quality-based Independent assessment-based
PS15	Tip et al.	Yes	No	Java	7	Yes	No	Yes	Type Constraint	Empirical-based
PS16	Soares et al.	Yes	No	AspectJ	7	Yes	No	Yes	Refactoring Safety Tool	Empirical-based Comparison-based
PS17	Overbey & Johnson	Yes	No	Fortran, PHP, BC	18	Yes	No	Yes	Differential Precondition Checking	Comparison-based
PS18	Soares et al.	Yes	No	Java	3	Yes	No	No	Overly Strong Preconditions Identification	Empirical-based
PS19	Soares et al.	Yes	No	Java	36	Yes	No	Yes	Refactoring Safety Tool Commit Message Analysis Manual Analysis	Comparison-based
PS20	Jonge & Visser	Yes	No	Java, Stratego, Mobl	1	Yes	No	No	Naming Binding Preservation	Quality-based
PS21	Noguera et al.	Yes	No	Java, AspectJ, XML	Not Mentioned	Yes	No	Yes	Refactoring Safety Tool	Formal specification-based
PS22	Thies et al.	Yes	No	Java	6	Yes	No	No	Refactoring Safety Tool	Empirical-based
PS23	Mongiovi et al.	Yes	No	Java,AspectJ	16	Yes	No	Yes	Refactoring Safety Tool	Comparison-based
PS24	Najafi et al.	No	Yes	UML		No	Yes	No	Annealing & Introduce Subtyping	Formal specification-based
PS25	HorpAacsi	Yes	No	Erlang	6	No	No	Yes	Decomposition & Schemes	Formal specification-based
PS26	Mongiovi et al.	Yes	No	Java	10	Yes	No	Yes	Overly Strong Preconditions Identification	Comparison-based
PS27	Chen et al.	Yes	No	Java	5	Yes	No	No	Refactoring Safety Tool	Empirical-based
PS28	Insa et al.	No	Yes	Erlang	Not Mentioned	No	No	Yes	Refactoring Safety Tool	Empirical-based

6.2. Implication for researchers:

- **Developing refactoring tools tuned towards safer refactoring.** As discussed in Section 5, our study sheds light on a number of desirable properties for refactoring tools (e.g., quality improvement, developer perception, automated testing, etc). Future researchers are encouraged to revisit the existing refactoring tools or build tools that help practitioners have more confidence in using the tools.

- **Exploring the potential of combining multiple behavior preservation strategies**. Our study shows that there are some behavior preservation strategies that have been evaluated using single or multiple refactoring operations, and some of these refactorings are applied using multiple strategies. Future researchers are advised to explore the potential of combining several behavior preservation approaches and use the approaches that would be useful in a given context according to a defined set of criteria.

7. Threats to Validity

In this section, the threats are discussed in the context of four types of threats of validity: internal validity, external validity, construct validity, and conclusion validity.

Internal validity: Obtaining a representative set of literature publications for this SLM can be viewed as a validity threat due to the search process. To minimize this threat, we followed the SLM guidelines proposed by [4, 5, 6]. We considered the related search terms and the main terms from research questions to construct the search string and select relevant articles. Further, we followed a four-stage study selection process and applied the inclusion and exclusion criteria in each stage as described in Section 3. Another threat is related to the limitation of the search terms and search engines which might lead to an incomplete set of literature publications. To limit this threat, we used carefully defined keywords and comprehensive academic search engines (i.e., Google Scholar and Scopus) that covers the main publisher venues.

External validity: The collected papers contain a significant proportion of academic works which forms an adequate basis for concluding findings that could be useful for academia. However, we cannot claim that the same behavior preservation approaches are used in industry. Also, our findings are mainly within the field of software refactoring. We cannot generalize our results beyond this subject.

Construct validity: Threats related to the construct validity are the suitability of the research questions and the categorization scheme used to extract the data. To mitigate these threats, the research questions and the categorization schemes were discussed among the authors.

Conclusion validity: Concerning the subjectivity of the assessment of the PS's, the primary studies were reviewed by at least two authors to mitigate bias in

data extraction. In case of disagreements, the researchers discussed these cases to reach consensus.

8. Conclusion

In this paper, we mapped and reviewed the body of knowledge on behavior preservation in software refactoring. We systematically reviewed 28 papers and classified them. This research sets out to aggregate, summarize, and discuss the practical approaches that ensure behavior-preserving refactoring transformations. Our main findings show that (1) code artifacts have the main focus in refactoring literature, (2) some refactoring types were studied more frequently than others, (3) several behavior preservation approaches proposed in the literature including the concepts and techniques that guarantee program correctness when dealing with refactoring activities, the automated analyses that are proposed, and the manual analysis approach, and (4) the majority of the PSs empirically evaluate their approaches. This existing research evaluates the correctness of the transformation and whether or not these approaches lead to a safe and trustworthy refactoring.

Lesson learned. Research around behavior preservation of software refactoring has mainly focused on precondition-based strategy. However, other techniques such as graph-based have potential and might be more effective for particular problems. Consequently, current and future research in this area should explore the suitability of each technique based on the context and the possibility of incorporating several strategies to ensure the correctness of program transformation. Further, current refactoring engines are limited to certain features. Future research should strive to implement a full-featured refactoring engine to increase developers' trust in refactoring tools.

Appendix A Primary Studies

List of accepted literature publications:

(PS1) D. Roberts, J. Brant, R. Johnson, A refactoring tool for smalltalk, Theory and Practice of Object systems 3 (4) (1997) 253–263

(PS2) T. Mens, N. Van Eetvelde, D. Janssens, S. Demeyer, Formalising refactorings with graph transformations, 2003, p. 69

(PS3) F. Tip, A. Kiezun, D. Bäumer, Refactoring for generalization using type constraints, in: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications, OOPSLA '03, ACM, New York, NY, USA, 2003, pp. 13–26. doi:10.1145/949305.949308.
URL http://doi.acm.org/10.1145/949305.949308

(PS4) A. Garrido, J. Meseguer, Formal specification and verification of java refactorings, in: 2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation, 2006, pp. 165–174. doi:10.1109/SCAM.2006.16

(PS5) R. Van Der Straeten, V. Jonckers, T. Mens, A formal approach to model refactoring and model refinement, Software & Systems Modeling 6 (2) (2007) 139–162. doi:10.1007/s10270-006-0025-9.
URL https://doi.org/10.1007/s10270-006-0025-9

(PS6) T. Massoni, R. Gheyi, P. Borba, Formal model-driven program refactoring, in: Proceedings of the Theory and Practice of Software, 11th International Conference on Fundamental Approaches to Software Engineering, FASE'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 362–376.
URL http://dl.acm.org/citation.cfm?id=1792838.1792873

(PS7) G. Soares, D. Cavalcanti, R. Gheyi, T. Massoni, D. Serey, M. Cornélio, Saferefactor-tool for checking refactoring safety

(PS8) N. Ubayashi, J. Piao, S. Shinotsuka, T. Tamai, Contract-based verification for aspect-oriented refactoring, in: 2008 1st International Conference on Software Testing, Verification, and Validation, IEEE, 2008, pp. 180–189

(PS9) M. Schäfer, T. Ekman, O. De Moor, Sound and extensible renaming for java, in: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, 2008, pp. 277–294

(PS10) G. Soares, R. Gheyi, T. Massoni, M. Cornélio, D. Cavalcanti, Generating unit tests for checking refactoring safety, in: Brazilian Symposium on Programming Languages, 2009, pp. 159–172

(PS11) N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Transactions on Software Engineering 35 (3) (2009) 347–367. doi:10.1109/TSE.2009.1

(PS12) M. Schäfer, O. De Moor, Specifying and implementing refactorings, in: Proceedings of the ACM international conference on Object oriented programming systems languages and applications, 2010, pp. 286–301

(PS13) G. Soares, R. Gheyi, D. Serey, T. Massoni, Making program refactoring safer, IEEE Software 27 (4) (2010) 52–57. doi:10.1109/MS.2010.63

(PS14) N. Tsantalis, A. Chatzigeorgiou, Identification of refactoring opportunities introducing polymorphism, Journal of Systems and Software 83 (3) (2010) 391–404

(PS15) F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, B. De Sutter, Refactoring using type constraints, ACM Transactions on Programming Languages and Systems (TOPLAS) 33 (3) (2011) 1–47

(PS16) G. Soares, D. Cavalcanti, R. Gheyi, Making aspect-oriented refactoring safer, in: Proceedings of the 15th Brazilian Symposium on Programming Languages, SBLP, Vol. 11, 2011, pp. 91–105

(PS17) J. L. Overbey, R. E. Johnson, Differential precondition checking: A lightweight, reusable analysis for refactoring tools, in: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 2011, pp. 303–312. doi:10.1109/ASE.2011.6100067

(PS18) G. Soares, M. Mongiovi, R. Gheyi, Identifying overly strong conditions in refactoring implementations, in: 2011 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 173–182. doi:10.1109/ICSM.2011.6080784

(PS19) G. Soares, R. Gheyi, E. Murphy-Hill, B. Johnson, Comparing approaches to analyze refactoring activity on software repositories, Journal of Systems and Software 86 (4) (2013) 1006 – 1022, sI : Software Engineering in Brazil: Retrospective and Prospective Views. doi:https://doi.org/10.1016/j.jss.2012.10.040.
URL http://www.sciencedirect.com/science/article/pii/S016412121200297X

(PS20) M. De Jonge, E. Visser, A language generic solution for name binding preservation in refactorings, in: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications, 2012, pp. 1–8

(PS21) C. Noguera, A. Kellens, C. De Roover, V. Jonckers, Refactoring in the presence of annotations, in: 2012 28th IEEE International Conference on Software Maintenance (ICSM), IEEE, 2012, pp. 337–346

(PS22) A. Thies, E. Bodden, Refaflex: Safer refactorings for reflective java programs, in: Proceedings of the 2012 International Symposium on Software Testing and Analysis, 2012, pp. 1–11

(PS23) M. Mongiovi, R. Gheyi, G. Soares, L. Teixeira, P. Borba, Making refactoring safer through impact analysis, Sci. Comput. Program. 93 (2014) 39–64. doi:10.1016/j.scico.2013.11.001.
URL http://dx.doi.org/10.1016/j.scico.2013.11.001

(PS24) M. Najafi, H. Haghighi, T. Z. Nasab, A set of refactoring rules for uml-b specifications, Computing and Informatics 35 (2) (2016) 411–440

(PS25) D. Horpácsi, J. Köszegi, Z. Horváth, Trustworthy refactoring via decomposition and schemes: A complex case study, in: VPT@ETAPS, 2017

(PS26) M. Mongiovi, R. Gheyi, G. Soares, M. Ribeiro, P. Borba, L. Teixeira, Detecting overly strong preconditions in refactoring engines, IEEE Transactions on Software Engineering 44 (5) (2018) 429–452. doi:10.1109/TSE.2017.2693982

(PS27) Z. Chen, H.-F. Guo, M. Song, Improving regression test efficiency with an awareness of refactoring changes, Information and Software Technology 103 (2018) 174–187

(PS28) D. Insa, S. Pérez, J. Silva, S. Tamarit, Behaviour preservation across code versions in erlang, Scientific Programming 2018

References

[1] M. Kim, T. Zimmermann, N. Nagappan, An empirical study of refactoring challenges and benefits at microsoft, IEEE Transactions on Software Engineering 40 (7) (2014) 633–649.

[2] D. Silva, N. Tsantalis, M. T. Valente, Why we refactor? confessions of github contributors, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, ACM, New York, NY, USA, 2016, pp. 858–870. doi:10.1145/2950290.2950305.
URL http://doi.acm.org/10.1145/2950290.2950305

[3] E. Murphy-Hill, C. Parnin, A. P. Black, How we refactor, and how we know it, IEEE Transactions on Software Engineering 38 (1) (2012) 5–18. doi:10.1109/TSE.2011.41.

[4] B. Kitchenham, S. Charters, Guidelines for performing systematic literature reviews in software engineering (2007).

[5] C. Wohlin, Guidelines for snowballing in systematic literature studies and a replication in software engineering, in: Proceedings of the 18th international conference on evaluation and assessment in software engineering, 2014, pp. 1–10.

[6] K. Petersen, R. Feldt, S. Mujtaba, M. Mattsson, Systematic mapping studies in software engineering, in: 12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12, 2008, pp. 1–10.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, d. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
URL http://dl.acm.org/citation.cfm?id=311424

[8] W. F. Opdyke, Refactoring object-oriented frameworks, Ph.D. thesis, Champaign, IL, USA, uMI Order No. GAX93-05645 (1992).

[9] T. Mens, T. Tourwe, A survey of software refactoring, IEEE Transactions on Software Engineering 30 (2) (2004) 126–139. doi:10.1109/TSE.2004.1265817.

[10] M. Zhang, T. Hall, N. Baddoo, Code bad smells: A review of current knowledge, J. Softw. Maint. Evol. 23 (3) (2011) 179–202. doi:10.1002/smr.521.
URL http://dx.doi.org/10.1002/smr.521

[11] M. Abebe, C.-J. Yoo, Trends, opportunities and challenges of software refactoring: A systematic literature review 8 (2014) 299–318.

[12] M. Misbhauddin, M. Alshayeb, Uml model refactoring: a systematic literature review, Empirical Software Engineering 20 (1) (2015) 206–251. doi:10.1007/s10664-013-9283-7.
URL https://doi.org/10.1007/s10664-013-9283-7

[13] J. A. Dallal, Identifying refactoring opportunities in object-oriented code: A systematic literature review, Information and Software Technology 58 (2015) 231 – 249. doi:https://doi.org/10.1016/j.infsof.2014.08.002.
URL http://www.sciencedirect.com/science/article/pii/S0950584914001918

[14] S. Singh, S. Kaur, A systematic literature review: Refactoring for disclosing code smells in object oriented software, Ain Shams Engineering Journaldoi:https://doi.org/10.1016/j.asej.2017.03.002.
URL http://www.sciencedirect.com/science/article/pii/S2090447917300412

[15] J. A. Dallal, A. Abdin, Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review, IEEE Transactions on Software Engineering PP (99) (2017) 1–1. doi:10.1109/TSE.2017.2658573.

[16] T. Mariani, S. R. Vergilio, A systematic review on search-based refactoring, Information and Software Technology 83 (2017) 14 – 34. doi:https://doi.org/10.1016/j.infsof.2016.11.009.
URL http://www.sciencedirect.com/science/article/pii/S0950584916303779

[17] A. A. B. Baqais, M. Alshayeb, Automatic software refactoring: a systematic literature review, Software Quality Journal 28 (2) (2020) 459–502.

[18] M. O. Cinnéide, Automated application of design patterns: a refactoring approach, Trinity College Dublin, 2001.

[19] V. Garousi, M. V. Mäntylä, A systematic literature review of literature reviews in software testing, Information and Software Technology 80 (2016) 195 – 216. doi:https://doi.org/10.1016/j.infsof.2016.09.002.
URL http://www.sciencedirect.com/science/article/pii/S0950584916301446

[20] H. , S. , A. , M. Abdollahi Azgomi, Uml model refactoring with emphasis on behavior preservation (2008) 125–.

[21] B. Kitchenham, P. Brereton, A systematic review of systematic review process research in software engineering, Information and Software Technology 55 (12) (2013) 2049 – 2075. doi:https://doi.org/10.1016/j.infsof.2013.07.010.
URL http://www.sciencedirect.com/science/article/pii/S0950584913001560

[22] F. Tip, A. Kiezun, D. Bäumer, Refactoring for generalization using type constraints, in: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications, OOPSLA '03, ACM, New York, NY, USA, 2003, pp. 13–26. doi:10.1145/949305.949308.
URL http://doi.acm.org/10.1145/949305.949308

[23] D. Roberts, J. Brant, R. Johnson, A refactoring tool for smalltalk, Theory and Practice of Object systems 3 (4) (1997) 253–263.

[24] T. Mens, N. Van Eetvelde, D. Janssens, S. Demeyer, Formalising refactorings with graph transformations, 2003, p. 69.

[25] A. Garrido, J. Meseguer, Formal specification and verification of java refactorings, in: 2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation, 2006, pp. 165–174. doi:10.1109/SCAM.2006.16.

[26] R. Van Der Straeten, V. Jonckers, T. Mens, A formal approach to model refactoring and model refinement, Software & Systems Modeling 6 (2) (2007) 139–162. doi:10.1007/s10270-006-0025-9.
URL https://doi.org/10.1007/s10270-006-0025-9

[27] T. Massoni, R. Gheyi, P. Borba, Formal model-driven program refactoring, in: Proceedings of the Theory and Practice of Software, 11th International Conference on Fundamental Approaches to Software Engineering, FASE'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 362–376.
URL http://dl.acm.org/citation.cfm?id=1792838.1792873

[28] G. Soares, D. Cavalcanti, R. Gheyi, T. Massoni, D. Serey, M. Cornélio, Saferefactor-tool for checking refactoring safety.

[29] N. Ubayashi, J. Piao, S. Shinotsuka, T. Tamai, Contract-based verification for aspect-oriented refactoring, in: 2008 1st International Conference on Software Testing, Verification, and Validation, IEEE, 2008, pp. 180–189.

[30] M. Schäfer, T. Ekman, O. De Moor, Sound and extensible renaming for java, in: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, 2008, pp. 277–294.

[31] G. Soares, R. Gheyi, T. Massoni, M. Cornélio, D. Cavalcanti, Generating unit tests for checking refactoring safety, in: Brazilian Symposium on Programming Languages, 2009, pp. 159–172.

[32] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Transactions on Software Engineering 35 (3) (2009) 347–367. doi:10.1109/TSE.2009.1.

[33] M. Schäfer, O. De Moor, Specifying and implementing refactorings, in: Proceedings of the ACM international conference on Object oriented programming systems languages and applications, 2010, pp. 286–301.

[34] G. Soares, R. Gheyi, D. Serey, T. Massoni, Making program refactoring safer, IEEE Software 27 (4) (2010) 52–57. doi:10.1109/MS.2010.63.

[35] N. Tsantalis, A. Chatzigeorgiou, Identification of refactoring opportunities introducing polymorphism, Journal of Systems and Software 83 (3) (2010) 391–404.

[36] F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, B. De Sutter, Refactoring using type constraints, ACM Transactions on Programming Languages and Systems (TOPLAS) 33 (3) (2011) 1–47.

[37] G. Soares, D. Cavalcanti, R. Gheyi, Making aspect-oriented refactoring safer, in: Proceedings of the 15th Brazilian Symposium on Programming Languages, SBLP, Vol. 11, 2011, pp. 91–105.

[38] J. L. Overbey, R. E. Johnson, Differential precondition checking: A lightweight, reusable analysis for refactoring tools, in: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 2011, pp. 303–312. doi:10.1109/ASE.2011.6100067.

[39] G. Soares, M. Mongiovi, R. Gheyi, Identifying overly strong conditions in refactoring implementations, in: 2011 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 173–182. doi:10.1109/ICSM.2011.6080784.

[40] G. Soares, R. Gheyi, E. Murphy-Hill, B. Johnson, Comparing approaches to analyze refactoring activity on software repositories, Journal of Systems and Software 86 (4) (2013) 1006 – 1022, sI : Software Engineering in Brazil: Retrospective and Prospective Views. doi:https://doi.org/10.1016/j.jss.2012.10.040.
URL http://www.sciencedirect.com/science/article/pii/S016412121200297X

[41] M. De Jonge, E. Visser, A language generic solution for name binding preservation in refactorings, in: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications, 2012, pp. 1–8.

[42] C. Noguera, A. Kellens, C. De Roover, V. Jonckers, Refactoring in the presence of annotations, in: 2012 28th IEEE International Conference on Software Maintenance (ICSM), IEEE, 2012, pp. 337–346.

[43] A. Thies, E. Bodden, Refaflex: Safer refactorings for reflective java programs, in: Proceedings of the 2012 International Symposium on Software Testing and Analysis, 2012, pp. 1–11.

[44] M. Mongiovi, R. Gheyi, G. Soares, L. Teixeira, P. Borba, Making refactoring safer through impact analysis, Sci. Comput. Program. 93 (2014) 39–64. doi:10.1016/j.scico.2013.11.001.
URL http://dx.doi.org/10.1016/j.scico.2013.11.001

[45] M. Najafi, H. Haghighi, T. Z. Nasab, A set of refactoring rules for uml-b specifications, Computing and Informatics 35 (2) (2016) 411–440.

[46] D. Horpácsi, J. Köszegi, Z. Horváth, Trustworthy refactoring via decomposition and schemes: A complex case study, in: VPT@ETAPS, 2017.

[47] M. Mongiovi, R. Gheyi, G. Soares, M. Ribeiro, P. Borba, L. Teixeira, Detecting overly strong preconditions in refactoring engines, IEEE Transactions on Software Engineering 44 (5) (2018) 429–452. doi:10.1109/TSE.2017.2693982.

[48] Z. Chen, H.-F. Guo, M. Song, Improving regression test efficiency with an awareness of refactoring changes, Information and Software Technology 103 (2018) 174–187.

[49] D. Insa, S. Pérez, J. Silva, S. Tamarit, Behaviour preservation across code versions in erlang, Scientific Programming 2018.

[50] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, D. Dig, Accurate and efficient refactoring detection in commit history, in: Proceedings of the 40th International Conference on Software Engineering, ACM, 2018, pp. 483–494.

[51] A. Peruma, M. W. Mkaouer, M. J. Decker, C. D. Newman, Contextualizing rename decisions using refactorings, commit messages, and data types, Journal of Systems and Software 169 (2020) 110704.

[52] E. A. AlOmar, A. Peruma, M. W. Mkaouer, C. Newman, A. Ouni, M. Kessentini, How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation, Expert Systems with Applications 167 (2021) 114176.

[53] N. Tsantalis, A. Ketkar, D. Dig, Refactoringminer 2.0, IEEE Transactions on Software Engineering.

[54] F. Baader, The description logic handbook: Theory, implementation and applications, Cambridge university press, 2003.

[55] W. M. McKeeman, Differential testing for software. 10 (1998) 100–107.

[56] T.-H. Dao, H. A. Le, N. T. Truong, An approach to analyzing execution preservation in java program refactoring, in: International Conference on Context-Aware Systems and Applications, Springer, 2016, pp. 101–110.

[57] M. Mongiovi, Safira: A tool for evaluating behavior preservation, in: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, 2011, pp. 213–214.

[58] J. Ratzinger, sPACE: Software Project Assessment in the Course of Evolution, Ph.D. thesis (2007).
URL http://www.infosys.tuwien.ac.at/Staff/ratzinger/publications/ratzinger_phd-thesis_space.pdf

[59] J. Ratzinger, T. Sigmund, H. C. Gall, On the relation of refactorings and software defect prediction, in: Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR '08, ACM, New York, NY, USA, 2008, pp. 35–38. doi:10.1145/1370750.1370759.
URL http://doi.acm.org/10.1145/1370750.1370759

[60] E. A. AlOmar, M. W. Mkaouer, A. Ouni, M. Kessentini, On the impact of refactoring on the relationship between quality attributes and design metrics, in: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, 2019, pp. 1–11.

[61] D. Zhang, B. Li, Z. Li, P. Liang, A preliminary investigation of self-admitted refactorings in open source software, 2018. doi:10.18293/SEKE2018-081.

[62] E. A. AlOmar, M. W. Mkaouer, A. Ouni, Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages, in: Proceedings of the 3nd International Workshop on Refactoring-accepted. IEEE, 2019.

[63] D. Silva, M. T. Valente, Refdiff: Detecting refactorings in version histories, in: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), 2017, pp. 269–279. doi:10.1109/MSR.2017.14.

[64] M. Vakilian, R. E. Johnson, Alternate refactoring paths reveal usability problems, in: Proceedings of the 36th international conference on software engineering, 2014, pp. 1106–1116.

[65] A. Bogart, E. A. AlOmar, M. W. Mkaouer, A. Ouni, Increasing the trust in refactoring through visualization, in: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, 2020, pp. 334–341.