IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received November 24, 2021, accepted December 15, 2021, date of publication January 19, 2022, date of current version February 2, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3144743

Automatizing Software Cognitive Complexity Reduction

RUBÉN SABORIDO, JAVIER FERRER, FRANCISCO CHICANO, AND ENRIQUE ALBA
ITIS Software, Universidad de Málaga, 29016 Málaga, Spain

Corresponding author: Rubén Saborido (rsain@uma.es)

This resarch has been supported by Universidad de Málaga (grants B1-2020_01 and B4-2019-05) and project PID2020-116727RB-I00 funded by MCIN/AEI /10.13039/501100011033. Rubén Saborido is recipient of a Juan de la Cierva grant FJC2018-038537-I funded by MCIN/AEI /10.13039/501100011033. Javier Ferrer is supported by a postdoc grant (DOC/00488) funded by the Andalusian Ministry of Economic Transformation, Industry, Knowledge and Universities. The views expressed are purely those of the writer and may not in any circumstances be regarded as stating an official position of the European Commission.

ABSTRACT Software plays a central role in our life nowadays. We use it almost anywhere, at any time, and for everything: to browse the Internet, to check our emails, and even to access critical services such as health monitoring and banking. Hence, its reliability and general quality is critical. As software increases in complexity, developers spend more time fixing bugs or making code work rather than designing or writing new code. Thus, improving software understandability and maintainability would translate into an economic relief over the total cost of a project. Different cognitive complexity measures have been proposed to quantify the understandability of a piece of code and, therefore, its maintainability. However, the cognitive complexity metric provided by SonarSource and integrated in SonarCloud and SonarQube is quickly spreading in the software industry due to the popularity of these well-known static code tools for evaluating software quality. Despite SonarQube suggests to keep method's cognitive complexity no greater than 15, reducing method's complexity is challenging for a human programmer and there are no approaches to assist developers on this task. We model the cognitive complexity reduction of a method as an optimization problem where the search space contains all sequences of Extract Method refactoring opportunities. We then propose a novel approach that searches for feasible code extractions allowing developers to apply them, all in an automated way. This will allow software developers to make informed decisions while reducing the complexity of their code. We evaluated our approach over 10 open-source software projects and was able to fix 78% of the 1,050 existing cognitive complexity issues reported by SonarQube. We finally discuss the limitations of the proposed approach and provide interesting findings and guidelines for developers.

INDEX TERMS Software quality, software maintenance, optimization, cognitive complexity.

I. INTRODUCTION

Most of the cost during software development is due to its maintenance [1], [2]. In complex software systems the time spent for validation could even be longer than the development time. Previous studies showed that debugging errors could be up to the 50% of the total cost of software projects [3]. This is due to the fact that software maintenance tasks are usually performed by hand instead of using automatic approaches.

Software metrics provide a quantitative basis for the development and validation of models of software development process. Information gained from metrics can be used in managing the development process in order to improve the reliability and quality of software products [4]. Cognitive informatics plays an important role in understanding the fundamental characteristics of software and cognitive complexity metrics are a good indicator for this [5]. A number of such measures have been proposed in the literature. However, there is no single metric which has the capability of measuring the complexity of a program based on multiple object-oriented concepts [6]. One of the most popular software metrics is the Cyclomatic Complexity, proposed by Thomas McCabe in 1976 [7]. This metric quantifies the control flow complexity of a piece of code and it has been extensively used in Object Oriented Programming (OOP) to compute the minimum number of test cases to cover a method. However, this metric is not adequate to quantify the understandability

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

```
// Cognitive complexity 7
int sumOfPrimes(int m)
{
 int total = 0;
 // +1
 B: for (int i=1; i<=m; ++i)
 {
  // +2 (1 by nesting)
  for (int j=2; j<i; ++j)
  {
   // +3 (2 by nesting)
   if (i % j == 0)
   {
    continue B; // +1
   }
  }
  total += i;
 }
 return total;
}
```

```
// Cognitive
    complexity 1
String getWords(int n)
{
 // +1
 switch (n)
 {
  case 1:
   return "one";
  case 2:
   return "two";
  case 3:
   return "three";
  default:
   return "> three";
 }
}
```

FIGURE 1. Two methods with equal Cyclomatic Complexity (4) but different SSCC.

and maintainability of a code and, therefore, its cognitive complexity. Recently, a novel cognitive complexity metric has been proposed and integrated in the well-known static code tools SonarCloud[1] and SonarQube,[2] an open-source service and platform, respectively, for continuous inspection of code quality, which are extensively used by developers and software factories today. This cognitive complexity metric, which we refer to as **SonarSource Cognitive Complexity (SSCC)**, has been defined as a measure of how hard the control flow of a method is to understand and maintain [8]. It breaks from the practice of using mathematical models to assess software maintainability. It starts from the precedents set by Cyclomatic Complexity, but uses human judgment to assess how structures should be counted and to decide what should be added to the model as a whole. The SSCC is given by a positive number which is increased every time a control flow sentence appear. Their nested levels also contribute to the SSCC of a method. Note that SonarQube suggests to keep methods' cognitive complexity no greater than 15, although this threshold can be set by the user to a different value.

As an introductory example, let us compare the `sumOfPrimes` and `getWords` methods, both shown in Fig. 1. Although they have equal Cyclomatic Complexity (4), it is intuitively obvious that the control flow of `sumOfPrimes` is more difficult to understand than that of `getWords`. The `sumOfPrime` method has a more complex control flow than that of `getWords`, mainly due to the nested loop and the `continue` statement. Thus, the cognitive effort required by developers to understand and maintain both codes is not the same, being `sumOfPrimes` much harder (SSCC = 7) than `getWords` (SSCC = 1).

Therefore, the cognitive complexity metric integrated in SonarQube yields method complexity scores which strike programmers as fairer relative assessments of understandability than have been available with previous models [8]. This assessment of understandability is also valid at the class level, just aggregating methods' cognitive complexity. Despite the fact that SSCC correlates with source code understandability in a meaningful way [9], software developers lack support to reduce the cognitive complexity of their code to a given threshold.

There are different refactoring operations to handle different tasks. However, Extract Method is the most versatile refactoring operation serving 11 different purposes [10]. In addition, identification of Extract Method refactoring opportunities for the decomposition of methods can be performed in an automatic way [11]. Due to its many uses [12], Extract Method has been recognized as the "Swiss army knife of refactorings" [10], [13]. It has also been recently used to reduce the complexity of code [14], [15] as we do in this paper. The main differences with already existing approaches are the following: they performed a more limited experimental validation, they do not impose any threshold for the cognitive complexity of the methods, and, most important, they are not able to generate sequences of feasible extractions. For example, in the project Knowage-Core, one of the projects analyzed as part of the case of study in this paper, there are more than 100 methods which require a sequence of code extractions to reduce their SSCC. However, previous approaches are not able to reduce the SSCC of those methods in a single execution.

We model the reduction of the SSCC to a given threshold as an optimization problem. The search space contains all feasible sequences of Extract Method refactoring opportunities. An optimal solution is one which reduces SSCC to the chosen threshold while minimizing the number of method extractions.[3] Note that the new extracted methods must be below the threshold too. We here propose an approach to reduce the SSCC of software projects in an automated way. We finally implement the proposed approach as a software tool for Java code and we apply it over 10 open-source software projects to reduce their SSCC. The developed tool will be available as an open-source project in a public repository.[4] To the best of our knowledge, the SSCC metric has not been properly validated as a cognitive complexity measure. We additionally perform a theoretical validation of this metric, which we include as an appendix of this paper.

We thus make the following contributions:

- Modeling the SSCC reduction to a given threshold as an optimization problem.
- Providing a software tool to reduce the SSCC of Java projects in an automated way.
- Validating the proposed approach over 10 real world open-source applications.
- Defining best practices to improve software readability and maintainability while benefiting the SSCC reduction task.
- Performing a theoretical validation of the SSCC metric.

The remainder of this paper is organized as follows. Section II discusses related work. Section III formulates the

[1]https://www.sonarcloud.org/
[2]https://www.sonarqube.org/

[3]We minimize the number of method extractions to reduce the number of modifications in the original code.
[4]https://github.com/rsain/SoftwareCognitiveComplexityReducer

SSCC reduction as an optimization problem. Section IV, introduces our approach for reducing the SSCC to a given threshold. In Section V we present the case of study and summarize the experimental setting for evaluating our proposal. Section VI provides the results of our experiments. Section VII discusses the limitations of our approach, reports interesting findings, and identifies open research gaps. Section VIII discusses the threats to the validity of our work. Finally, Section IX presents conclusions and future work.

II. RELATED WORK

Probably the oldest and most intuitively obvious notion of software complexity is the number of statements in the program, or the statement count. However, a large number of software complexity measures have been proposed in the past. In the 70s, the number of program statements, McCabe's cyclomatic number [7], Halstead's programming effort [16], and the Knot measure [17] were the most frequently cited measures. In the 90s, Douce *et al.* introduced a set of metrics that help in calculating the complexity of a given system or program code based on the object-oriented concepts such as the object and class [18]. All those metrics were based on the spatial abilities which measure the complexity by calculating the distances between the program elements in the code.

In 2003, Shao and Wang proposed cognitive complexity as a new measure of the cognitive and psychological complexity of software by examining the cognitive weights of basic control structures (BCS) of software. Based on this approach a new concept of Cognitive Functional Size (CFS) of software was developed [19]. Cognitive weights are degree of difficulty or relative time and effort required for comprehending a given piece of software. In 2006, Misra *et al.* proposed the modification in CFS measure by taking into account the total occurrence of operators and operands and all internal BCS [20]. The same year, Misra proposed the Cognitive Weight Complexity Measure (CWCM) complexity measure which is also based on cognitive weights [21]. Then, Kushwaha and Misra framed different cognitive complexity metrics with the goal of aiding in increasing the reliability of software product during the development lifecycle [4].

In 2007, Misra S. and Misra A. K. compared cognitive complexity measures in terms of nine properties [22]. Then, Misra proposed an improved cognitive complexity measure named Cognitive Program Complexity Measure (CPCM) which establishes a relation between total number of inputs and outputs, cognitive weights, and cognitive complexity [23]. The same year, Misra proposed an object-oriented complexity metric which calculates the complexity of a class at method level [24]. Later, in 2008, Misra *et al.* proposed a metric that considers internal attributes which directly affect the complexity of software: number of lines, total occurrence of operators and operands, number of control structures, and function calls (coupling) [25]. The same year, Misra and Akman proposed a new complexity metric based on cognitive informatics for object-oriented code covering cognitive complexity of the system, method complexity, and complexity due to inheritance together [26].

Few years later, in 2011, Misra *et al.* proposed a cognitive complexity metric for evaluating design of object-oriented code. The proposed metric is based on the inheritance feature of the object-oriented systems. It calculates the complexity at method level considering internal structure of methods, and also considers inheritance to calculate the complexity of class hierarchies [27]. In 2012, Misra *et al.* proposed a suite of cognitive metrics for evaluating complexity of object-oriented codes [28]. All the metrics are critically examined through theoretical and empirical validation processes. The same year, Misra *et al.* also proposed a framework for the evaluation and validation of software complexity measure. This framework is designed to analyse whether or not software metric qualifies as a measure from different perspectives [29].

In 2016, Haas and Hummel addressed the problem of finding the most appropriate refactoring candidate for long methods written in Java. The approach determines valid refactoring candidates and ranks them using a scoring function that aims to improve readability and reduce code complexity [14]. Later that year, Wijendra and Hewagamage proposed a cognitive complexity metric which determines the amount of information inside the software through cognitive weights and the way of information scattering in terms of Lines of Code (LOC) [30]. In this paper, authors also analyzed how the proposed cognitive complexity calculation can be automated. The same year, Crasso *et al.* presented a software metric to assess cognitive complexity in object-oriented systems developed in the Java language [31]. The proposed metric is based on a characterization of basic control structures present in Java systems. Authors also provided several algorithms to compute the metric and introduced their materialization in the Eclipse IDE. Finally, the applicability of the tool was shown by illustrating the metric in the context of 10 real world Java projects.

In 2017, Rabani and Maheswaran discussed and analyzed classical and modern metrics of software cognitive complexity [5]. The same year, Misra *et al.* identified the features and advantages of the existing software cognitive complexity metrics [32]. They also performed a comparative analysis based on some selected criteria. The results showed that there is a similar trend in the output obtained from the different measures when they are applied to different examples.

In 2018, Misra *et al.* presented an updated suite of cognitive complexity metrics that can be used to evaluate object-oriented software projects [33]. The metrics suite was evaluated theoretically using measurement theory and Weyuker's properties and practically using Kaner's framework [34]. The same year, SonarSource introduced cognitive complexity as a new metric for measuring the understandability of any given piece of code [8]. This paper investigated developers' reaction to the introduction of cognitive complexity in the static code analysis tool service SonarCloud. In an analysis of 22 open-source projects, they assessed whether a development team 'accepted' the proposed metric based on whether they fixed

code areas of high cognitive complexity as reported by the tool. They found that the metric had a 77% acceptance rate among developers.

In 2019, Kaur and Mishra conducted an experimental analysis in which the software developer's level of difficulty in comprehending the software (the cognitive complexity) was theoretically computed and empirically evaluated for estimating its relevance to actual software change [35]. This study validated a cognitive complexity metric as a noteworthy measure of version to version source code change. Also in 2019, Alqadi proposed novel metrics to compute the cognitive complexity of code slices [36]. Empirical investigation into how cognitive complexity correlates with defects in the version histories of three open-source systems was performed. The results showed that the increase of cognitive complexity significantly increases the number of defects in 93% of the cases. The same year, Hubert proposed an approach to fully automate the extract method refactoring task ranking refactoring opportunities according to a scoring function which takes into account software cognitive complexity [15].

Recently, in 2020, Jayalath and Thelijjagoda proposed a new metric to evaluate the complexity of object-oriented programs based on the influence of previous object-oriented metrics and some disregarded factors in calculating the complexity [6]. The same year, Muñoz Barón *et al.* conducted a systematic literature search to obtain data sets from studies which measured code understandability and found that cognitive complexity integrated in the well-known static code analysis tool service SonarCloud positively correlates with comprehension time and subjective ratings of understandability [9].

Although existing approaches can indirectly reduce software cognitive complexity, they are not able to automatically reduce methods cognitive complexity to a given threshold. Our proposal is novel because we (i) model the software cognitive complexity reduction to a given threshold as an optimization problem and (ii) provide a tool that generates and applies a sequence of feasible extractions to reduce the SonarSource Cognitive Complexity of Java projects.

III. PROBLEM DEFINITION AND MOTIVATION

SonarCloud and SonarQube compute cognitive complexity of a method as the sum of two components that we call the *inherent component* and the *nesting component*. The *inherent component* depends on the presence of certain control flow structures and complex expressions (like decisions combining several conditional expressions). When a control flow structure or complex expression is found it contributes +1 to the *inherent component*. The *nesting component* depends on the depth that a certain control flow structure is in the code with respect to the root node (e.g. a method declaration). This depth is the contribution to the *nesting component*. Let s_i and e_i be the start and end offset (in characters) of the ith sequence of sentences of a method in its source file. We consider that ith sequence is *nested* in the jth sequence, denoted with $i \rightarrow j$, when $[s_i, e_i] \subset [s_j, e_j]$. We say that the ith sequence is *in conflict with* the jth sequence, denoted with $i \nleftrightarrow j$, when i and j are not nested one in the other and $[s_i, e_i] \cap [s_j, e_j] \neq \emptyset$.

The SSCC at method level can be reduced to a threshold applying Extract Method refactorings: extracting as a new method in the same class sequences of sentences (i.e., lines of code). However, this task is not straightforward for software developers due to the following reasons:

1) The number of different Extract Method refactoring opportunities l is bounded by $\binom{n}{2} = \frac{n \cdot (n-1)}{2}$, where n is the number of sentences of the method.[5]
2) Two code extractions cannot be applied simultaneously if they are in conflict.
3) Extract method refactoring opportunities are not applicable when they introduce compilation errors or the SSCC of the extracted code cannot be reduced to the threshold.
4) More than one Extract Method refactoring could be required to reduce the SSCC of a method.

Based on the previous, we define the method cognitive complexity reduction task as an optimization problem which asks "*What is the optimal sequence of extract-method refactoring to apply in order to reduce the SSCC of the original method to/below a given threshold?*". Thus, a solution to this problem is a sequence of code extractions which is bounded by 2^l (all possible combinations of Extract Method refactorings).

A. CHALLENGES OF REDUCING SOFTWARE COGNITIVE COMPLEXITY

Fig. 2 shows a running example to illustrate the difficulties developers face when reducing the SSCC of a method. Note that some code has been replaced by "..." due to space limitations, but the whole code is accesible in the following URL.[6] This method has SSCC 46 and SonarQube suggests to reduce it to 15 in order to improve the understandability and maintainability of the method. As shown, there are 37 statements and the upper bound of Extract Method refactoring opportunities is $\binom{37}{2} = 666$. However, we have checked computationally that there are only 28 applicable code extractions. After analyzing the method, a developer who faces this cognitive complexity reduction task could realize that the optimal solution is a sequence of three Extract Method refactorings. Therefore, one would need to evaluate all possible sequences of one, two, and three Extract Method operations totaling $\binom{28}{1} + \binom{28}{2} + \binom{28}{3} = 28 + 378 + 3,276 = 3,682$ solutions. Although this number of solutions is much smaller than the theoretical upper bound of all possible sequences of extractions explained in Section III (which is $2^{28} \approx 268$ million solutions), it is still unmanageable for developers without an automated approach.

[5] Combination of n sentences taken two at a time without repetition. Note that those two sentences determine the beginning and ending of a code extraction.

[6] https://github.com/KnowageLabs/Knowage-Server/blob/master/knowage-core/src/main/java/it/eng/knowage/api/dossier/DocumentExecutionWorkForDoc.java

```
1  //Cognitive complexity 46
2  public String addParametersToServiceUrl(...) throws
       ...
3  {
4   List<BIObjectParameter> drivers = ...;
5   // +1
6   if (drivers != null) {
7    List<Parameter> parameter = ...;
8    // +2 (1 by nesting)
9    if (drivers.size() != parameter.size()) {
10     throw new SpagoBIRuntimeException("There are...");
11   }
12   Collections.sort(drivers);
13   ParametersDecoder decoder = new ParametersDecoder();
14   // +2 (1 by nesting)
15   for (BIObjectParameter biObjectParameter : drivers)
16   {
17    boolean found = false;
18    String value = "";
19    String paramName = "";
20    // +3 (2 by nesting)
21    for (Parameter templateParameter : parameter) {
22     // +4 (3 by nesting)
23     if (templateParameter.getType().equals("dynamic")
24     {
25      // +5 (4 by nesting), +1 (logical expression)
26      if (templateParameter.getValue() != null && ...)
           {
27       value = templateParameter.getValue();
28       // +6 (5 by nesting), +1 (logical expression)
29       if (... && value.contains("STRING"))
30        value.replaceAll("'", "");
31
32       // +6 (5 by nesting)
33       if (...) {
34        paramName = templateParameter.getUrlName();
35        serviceUrlBuilder.append(...);
36        serviceUrlBuilder.append(...);
37        found = true;
38        break;
39       }
40      }
41     }
42     // +1
43     else {
44      // +5 (4 by nesting)
45      if (biObjectParameter.getParameterUrlName...) {
46       serviceUrlBuilder.append(...);
47       value = templateParameter.getValue();
48       paramName = templateParameter.getUrlName();
49       // +6 (5 by nesting)
50       if (templateParameter.getUrlNameDescription...)
51       {
52        throw new SpagoBIRuntimeException("...");
53       }
54       serviceUrlBuilder.append(...);
55       found = true;
56       break;
57      }
58     }
59    }
60    paramMap.put(paramName, value);
61    // +3 (2 by nesting)
62    if (!found) {
63     throw new SpagoBIRuntimeException("...");
64    }
65   }
66  }
67  return serviceUrlBuilder.toString();
68 }
```

FIGURE 2. Method with SSCC 46 used as running example along the paper. Reducing SSCC to 15 requires to evaluate more than 3,000 Extract Method refactoring opportunities. This method belongs to the Knowage-core software project.

IV. COGNITIVE COMPLEXITY REDUCER APPROACH

We propose a SSCC reducer approach consisting in a solver method implementing an automatic algorithm that takes as input the path to the software project to process and the cognitive complexity threshold (τ). Then, for each method with SSCC greater than τ, it searches for sequences of applicable Extract Method refactoring operations. Finally, it shows the changes to perform to each method and apply them all at once in an automated way.

In order to search for Extract Method refactoring opportunities in a method, our approach generates its corresponding Abstract Syntax Tree (AST). Second, it parses the AST and annotates different properties[7] in each node: its contribution to the SSCC of the method, the accumulated value of the *inherent component* (ι), the accumulated value of the *nesting component* (ν), the number of elements contributing to the *nesting component* of the SSCC of the node (μ), and its absolute nesting level (λ). Note that λ is 0 when no nesting exists in the target piece of software. Third, the approach processes the annotated AST to compute the list of consecutive sentences contributing to the SSCC of the method. This is done to obtain Extract Method refactoring opportunities. Although sentences contributing to the SSCC must be part of code extractions, it is also necessary to consider single statements even if they do not contribute to the value of this metric. The inclusion of statements of this kind could suppose that the extraction is feasible or not. For example when several arithmetic operations are needed to compute a result, if all operations are not included in the extraction, the refactoring probably is not possible because only one variable could be returned. Once the approach identifies Extract Method refactoring opportunities, it checks if the extractions are applicable. This is done with the help of refactoring tools which are able to check pre-conditions, post-conditions, and apply the corresponding operation over the source code.

A. COGNITIVE COMPLEXITY REDUCER TOOL IMPLEMENTATION

We propose a Java cognitive complexity reducer tool as an Eclipse application. The goal is to provide the necessary means for generating an Eclipse product that can be run from the operating system command-line as a standalone executable, without the need for opening Eclipse for running. This is particularly useful if, for instance, one needs to integrate it in their current development workflow (e.g., using continuous integration). We got this idea from the jDeodorant project,[8] an Eclipse plug-in that detects design problems in Java software and recommends appropriate refactorings to resolve them.

The developed tool takes as input (i) a SonarQube server URL, (ii) the path to the software project to process, (iii) the cognitive complexity threshold (τ), and (iv) a stopping criteria (a number of Extract Method refactoring evaluations). Then, it runs SonarQube to perform an analysis of the project and get all existing cognitive complexity issues. Finally, for each method with SSCC greater than τ, it searches for Extract Method refactoring opportunities. In order to do this, the tool first generates and processes the AST associated to the method declaration as explained in the previous section. Then, it enumerates sequences of applicable Extract Method refactorings while the given stopping criteria is not met. The tool uses the Extract Method refactoring operation provided

[7] These are used to compute the SSCC of extracted methods.
[8] https://github.com/tsantalis/JDeodorant

by the Java Development Toolkit (JDT) of Eclipse to test the feasibility of code extractions programmatically. Finally, the tool chooses the best sequence of method extractions found during the search: the one that reduces the SSCC to (or below) the threshold and minimizes the number of method extractions. If wished, the tool applies the required code extractions in an automated way by using the Extract Method refactoring provided by JDT.

The tool internally generates for each method under processing what we name the *conflicts graph*. A *conflicts graph* is a directed graph where vertices are applicable extractions and edges represent nested sequences of statements (i.e., if an edge targets j from i, then $i \rightarrow j$). The tool labels vertices in the *conflicts graph* as $[s_i, e_i](CC_i, \iota_i, \nu_i, \mu_i, \lambda_i)$, where s_i and e_i refer to the start and end offset (in characters in the source file) of the ith extraction. Red edges connect conflict vertices in the *conflicts graph* (i.e., $i \nleftrightarrow j$). Note that two vertices in conflict cannot be both selected for extraction in the same sequence. The root in a *conflicts graph* is a special vertex representing the whole body of the method. *Conflicts graphs* are used when searching for applicable Extract Method refactorings and to compute the impact of code extractions when reducing the SSCC of a method. Fig. 3 shows the *conflicts graph* of the running example whose source code is shown in Fig. 2. As shown, there are 28 extractable nodes plus the root node which is located in the left lower corner. In addition, there are 35 black edges that represent nested sequences of statements and 44 red edges that represent 22 pairs of nodes in conflict.

V. CASE STUDY

In this section we describe the study we conduct to evaluate the proposed approach when reducing the SSCC of 10 open-source projects. Next, we detail the objects of study. Then, we report the experimental setup used to conduct the experiments.

A. OBJECTS OF STUDY

We used the GitHub REST API to create calls to get repositories from GitHub satisfying two conditions: Java applications using Apache Maven as software project management. We choose Maven as software management because it eases the execution of SonarQube analysis via a regular Maven goal. We ended up selecting a diverse set of 10 open-source projects: two popular frameworks for multi-objective optimization, five platform components to accelerate the development of smart solutions, and three popular open-source projects with more than 10,000 stars and forked more than 900 times. Table 1 shows these projects and some software metric values. In order to ease the replication of the study, for each software project we also show its abbreviated commit hash in GitHub.

The simplest open-source project is QueryExecution: it contains 53 methods and six classes, summing up 1,013 lines of code. Despite the low number of methods in comparison to other open-source projects, 6 over 53 (11%) of the methods

TABLE 1. Case study projects metrics: name of the open-source project (its abbreviated commit hash), number of classes, number of methods, lines of code, and number of cognitive complexity issues reported by SonarQube, respectively.

Project (Commit)	#Classes	#Methods	LOC	#CC issues
ByteCode (55bfc32)	301	1,350	23,071	57 (4%)
CyberCaptor (b6b1f10)	85	784	17,023	37 (5%)
FastJson (93d8c01e9)	256	2,039	43,644	230 (11%)
Fiware-Commons (f83b342)	28	155	1,325	4 (3%)
IoTBroker (98eeceb)	79	646	7,940	10 (2%)
Jedis (cfc227f7)	295	1,917	16,566	3 (<1%)
jMetal (e6baf75aa)	610	3,327	43,298	63 (2%)
Knowage-core (dfed28a869)	1,093	6,967	149,137	558 (8%)
MOEA-framework (223393fd)	506	2,939	33,888	82 (3%)
QueryExecution (c032e5a)	6	53	1,013	6 (11%)

of QueryExecution have SSCC greater than 15 (the default threshold). Although this project looks simple, and, therefore, easy to maintain, reducing the SSCC of these six methods is not straight forward. For instance, for the method `getDBIds` SonarQube suggests to reduce its SSCC from 41 to 15. However, there are several refactoring opportunities that can be applied to get this done. Conversely, Knowage-core is the most complex project in our case study: it contains 6,967 methods and 1,093 classes, summing up 149,137 lines of code. Even for a senior developer, maintaining this ecosystem is complicated and prone to errors. SonarQube reports 558 cognitive complexity issues for this project, i.e., 8% of the methods in the project have SSCC greater than 15. Reducing the SSCC of these 558 methods would be time consuming and prone to errors when done manually.

We validate the proposed cognitive complexity reduction tool over the 10 open-source projects shown in Table 1. In total, these projects have 1,050 cognitive complexity issues. The goal of the study is to validate if the proposed approach is able to reduce the number of cognitive complexity issues existing on these projects. In addition, we want to uncover how many extractions are needed, how many lines of codes are extracted, and how many parameters new extracted methods have when reducing the SSCC of methods.

B. ALGORITHMS

We use an exhaustive search as resolution technique because it is conceptually simple and effective. It generates possible sequences of code extractions and assures the optimal one when all combinations can be generated. We want to keep the resolution technique simple to focus more on the problem and not in the resolution process.

The algorithm generates an exhaustive list of refactoring candidates first. To get this list, the source code is transformed into a block structure which contains structural information. After that, the algorithm starts to enumerate all possible code extractions in a recursive way with the help of a stack structure. The way the elements are introduced in the stack determines two variants of the algorithm: Exhaustive Search-Long Sequences First (ES-LSF) and Exhaustive Search-Short Sequences First (ES-SSF). The former is aimed at exploring as many consecutive statements as possible in a single extraction first. In contrast, the latter is aimed at exploring short sequences of statements first. We propose these two different

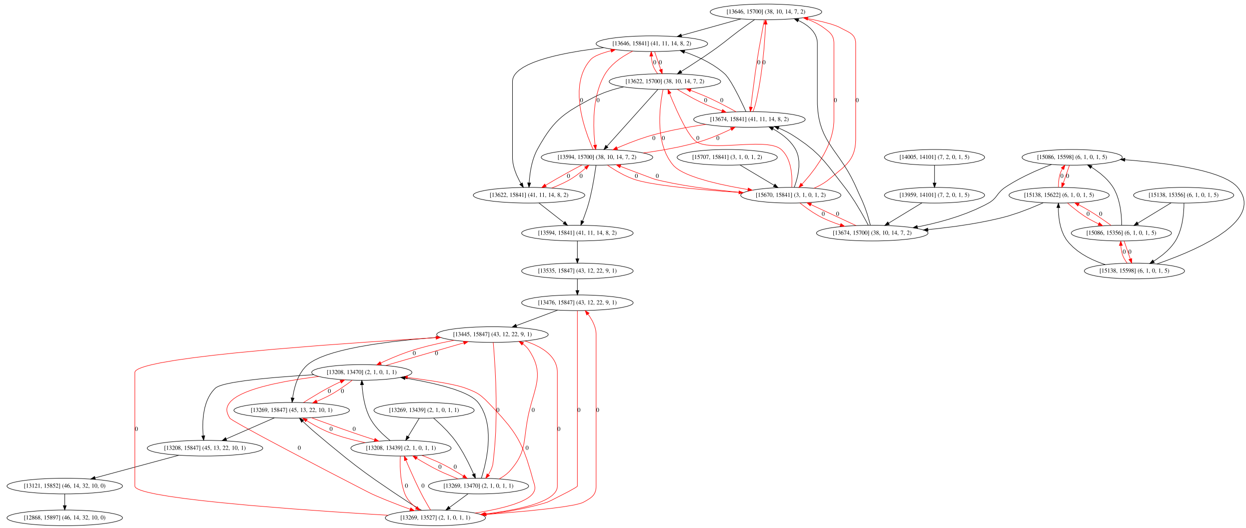

FIGURE 3. ***Conflicts graph*** **of the running example** `addParametersToServiceUrl` **which belongs to the Knowage-core software project.**

ways of exploring the search space because we set a number of evaluations as stopping criteria. If no stop condition is set, both variants must return an optimal solution.

C. EXPERIMENTAL SETUP

We conducted the experiments in a laptop Dell XPS 15 9560 with 4 × Intel® Core i7-7700HQ CPU @ 2.80GHz and 16 GiB of RAM, running the operating system Windows 10 Pro. We used SonarQube version 7.2 and the Eclipse IDE version 2020-06 (4.16.0). We set the cognitive complexity threshold to the default value proposed by SonarQube ($\tau = 15$). AST processing and Extract Method refactorings were performed through JDT version 3.16.0. All graph generation in our tool has been developed using the jGraphT library, a Java library of graph theory data structures and algorithms.[9] In order to check if the observed differences in the results of ES-LSF and ES-SSF are statistically significant, we applied the non-parametric Mann–Whitney–Wilcoxon test with a confidence level of 95% (p-value < 0.05).

VI. RESULTS

Table 2 reports the number of cognitive complexity issues, the number (and percentage) of cognitive complexity issues fixed, and the number and percentage of cognitive complexity issues that keep unfixed, respectively, for the projects under study.

As shown, the proposed approach is able to fix, on average, 78% of the cognitive complexity issues on these projects. Therefore, our approach is able to fix most cognitive complexity issues in most of the projects under study. However, 288 methods out of 1,050 (27%) has no solution since there are not applicable Extract Method refactorings. The reason is that most of these methods use multiple `return` statements

TABLE 2. **Number of cognitive complexity issues for projects under study: initial number of cognitive complexity issues (`Total`), number of issues that the proposed approach was able to fix (`Fixed`) and those where it was not (`Unfixed`).**

Project	Total	Fixed (%)	Unfixed (%)
ByteCode	57	43 (75%)	14 (25%)
CyberCaptor	37	28 (76%)	9 (24%)
FastJson	230	113 (49%)	117 (51%)
Fiware-Commons	4	3 (75%)	1 (25%)
IoTBroker	10	9 (90%)	1 (10%)
Jedis	3	2 (67%)	1 (33%)
Jmetal	63	54 (86%)	9 (14%)
Knowage-core	558	432 (77%)	126 (23%)
MOEA	82	72 (88%)	10 (12%)
QueryExecution	6	6 (100%)	0 (0%)

and loops containing multiple `break` or `continue` statements. These kind of statements prevent the extraction of any piece of code contributing to the SSCC of the method.

Table 3 summarizes the combined results of our experiments. The first column shows the name of the different variants of the exhaustive algorithm implemented in our tool. The second column indicates whether found solutions are feasible or not. Note that a solution is a sequence of Extract Method refactorings. We define a solution as feasible when the original method and all the new extracted methods have a SSCC no greater than τ. In other case, the solution is unfeasible. Note that the best solution is that one which minimizes the number of method extractions. The remaining columns show some aggregated function values (min, max, mean, standard deviation, and sum) for different metrics. Next, there are four blocks of metrics. The first one (columns 4-11) is devoted to SSCC related metrics: `iniCC` is the initial cognitive complexity of the original methods (always above the threshold), `extrac` is the number of Extract Method refactorings proposed by the best solution, `reducCC` is the reduction on the cognitive complexity of the original methods, `minReduc` is the minimum reduction for a single extraction, `maxReduc` is the maximum reduction for a single extraction,

[9] https://jgrapht.org/

TABLE 3. Results using ES-LSF and ES-SSF algorithms to reduce the SSCC of methods in the 10 analysed projects. The results are divided into feasible solutions and unfeasible solutions. The mean value is highlighted when there are statistical differences between ES-LSF and ES-SSF.

algorithm	feasible	stats	iniCC	extrac	reducCC	minReduc	maxReduc	avgReduc	totalReduc	finalCC	minLOC	maxLOC	avgLOC	totalLOC	minParams	maxParams	avgParams	totalParams	Time (ms)
ES-LSF	False	Min	20.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	4.00	2.60	4.00	0.00	0.00	0.00	0.00	337
		Max	582.00	10.00	103.00	93.00	103.00	93.00	169.00	528.00	208.00	208.00	208.00	208.00	9.00	17.00	10.00	41.00	725,457
		Mean	82.70	3.15	31.15	7.74	22.05	13.64	35.24	51.55	**12.44**	36.84	**22.15**	56.35	2.10	4.15	3.07	9.01	88,473.14
		SD	67.44	1.63	17.46	11.22	17.11	12.01	24.56	63.04	20.61	28.88	20.48	34.34	1.68	2.39	1.76	5.94	114,130.69
		Sum	23,320	887	8,784	2,184	6,219	3,847	9,937	14,536	3,509	10,390	6,245	15,892	591	1,169	865	2,542	24,949,426
	True	Min	16.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	0.00	0.00	0.00	221
		Max	86.00	5.00	72.00	44.00	63.00	44.50	120.00	15.00	148.00	148.00	148.00	154.00	15.00	17.00	15.00	27.00	471,871
		Mean	25.63	1.36	13.73	**9.32**	12.19	**10.64**	14.63	11.90	**16.84**	**22.59**	**19.46**	26.31	**2.97**	3.44	3.19	4.38	61,005.30
		SD	10.19	0.73	10.66	6.76	9.42	7.27	13.58	3.68	15.72	19.65	15.87	23.94	1.88	2.20	1.93	3.67	76,496.26
		Sum	19,452	1,036	10,418	7,073	9,254	8,076	11,106	9,034	12,781	17,144	14,772	19,968	2,255	2,614	2,425	3,327	46,303,022
ES-SSF	False	Min	20.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	3.00	1.60	3.00	0.00	0.00	0.00	0.00	500
		Max	582.00	10.00	103.00	93.00	103.00	93.00	169.00	534.00	185.00	185.00	185.00	193.00	9.00	17.00	9.00	41.00	786,305
		Mean	82.89	3.31	30.79	6.69	21.02	12.57	34.39	52.10	9.91	33.56	19.26	52.26	1.84	4.01	2.84	8.89	80,955.65
		SD	67.59	1.69	17.10	10.58	16.59	11.38	23.85	63.26	16.09	25.70	16.54	31.68	1.54	2.27	1.60	5.91	103,199.81
		Sum	23,210	928	8,621	1,872	5,885	3,521	9,630	14,589	2,775	9,397	5,393	14,633	516	1,123	795	2,489	22,667,581
	True	Min	16.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	0.00	0.00	0.00	129
		Max	86.00	30.00	72.00	44.00	63.00	44.00	120.00	15.00	98.00	134.00	98.00	145.00	15.00	17.00	15.00	84.00	750,624
		Mean	25.71	1.50	13.10	8.30	11.26	9.62	13.81	**12.60**	14.24	19.91	16.74	23.81	2.74	3.27	3.00	4.40	62,112.72
		SD	10.41	1.44	10.66	6.31	8.94	6.74	13.12	3.23	13.44	17.36	13.62	21.97	1.82	2.09	1.84	4.73	83,622.72
		Sum	19,562	1,141	9,971	6,315	8,566	7,319	10,507	9,591	10,833	15,151	12,740	18,117	2,087	2,488	2,281	3,349	47,267,778

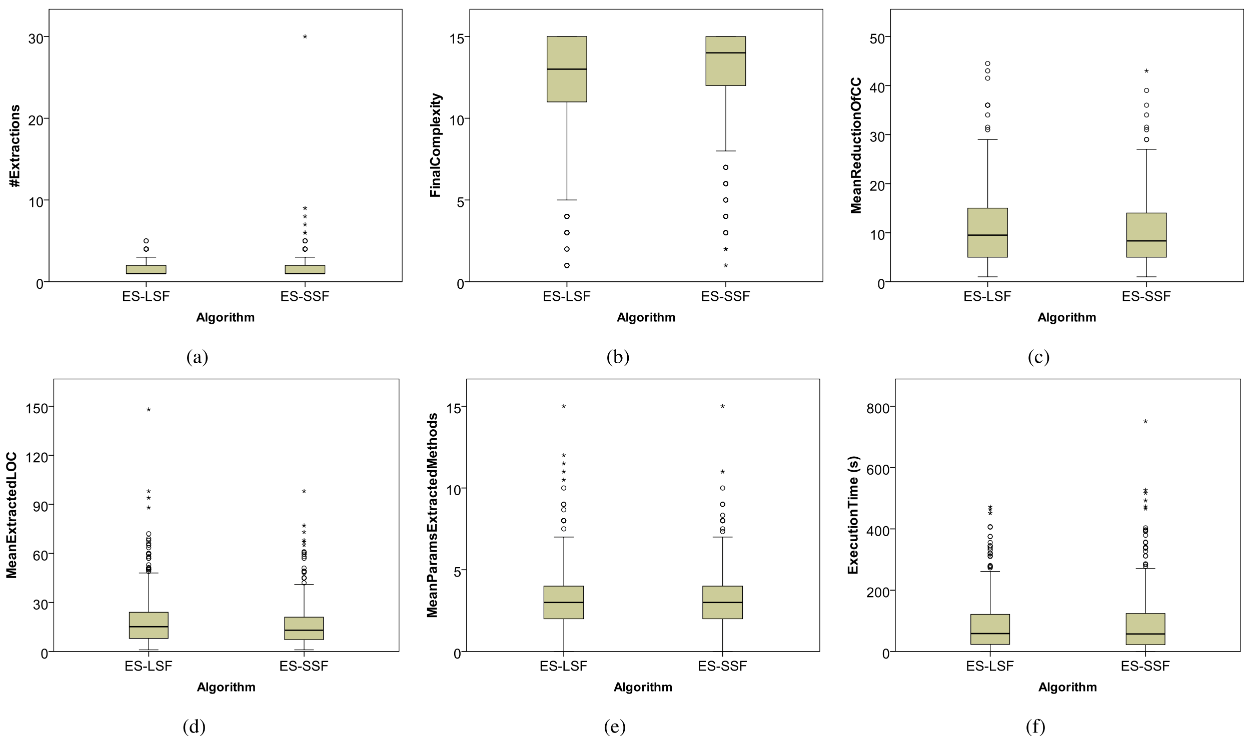

FIGURE 4. Boxplots of most interesting metrics for the two algorithms considering only feasible solutions.

`avgReduc` is the mean (average) reduction considering all extractions of the best solution, `totalReduc` is the sum of the reductions of all extractions of the best solution, and `finalCC` is the SSCC of the original methods after applying the sequence of Extract Method refactorings determined by the best solution. The second block (columns 12-15) shows LOC metrics, the third block (columns 16-19) provides information about the parameters involved in the extracted methods, and, finally, the last block shows the execution time in milliseconds.

ES-LSF and ES-SSF found 291 and 289 unfeasible solutions, respectively (i.e., they were able to reduce the SSCC of those methods but above τ). Interestingly, those algorithms found 759 and 761 feasible solutions for existing cognitive complexity issues, respectively. The slight difference between the two algorithms is due to the way they explore the search space.

Next, we focus on feasible solutions. Existing methods in the source code require, on average, more than one code extraction to reduce its SSCC. These code extractions reduce the SSCC of original methods below τ and extract around 15-20 lines of code from their original location into new methods. ES-LSF prioritizes during the search long code extractions while ES-SSF prefers short ones. Based on this, we expect that, on average, solutions found by ES-LSF extract portions of code with higher SSCC, more lines of code, and need more parameters for the new extracted methods, than solutions found by the ES-SSF algorithm. This is supported by statistical differences as shown in Table 4.

Fig. 4 shows boxplots of different metrics for feasible solutions. In Fig. 4(a) we can appreciate that there exist solutions with a high number of extractions for the ES-SSF algorithm. In contrast, the ES-LSF algorithm obtains solutions with lower number of code extractions. As commented previously,

TABLE 4. Mann–Whitney–Wilcoxon statistical test results (p-value) for feasible solutions. When no significant differences exist in a metric, we use the symbol ⊙. When a metric for ES-LSF is greater than for ES-SSF, we use the symbol ▲.

Metric	Test result
iniCC	⊙ 0.985
extrac	⊙ 0.391
reducCC	⊙ 0.293
minReduc	▲0.026
maxReduc	⊙ 0.104
avgReduc	▲0.039
totalReduc	⊙ 0.285
finalCC	▽ 0.012
minLOC	▲ 0.004
maxLOC	▲ 0.043
avgLOC	▲ 0.007
totalLOC	⊙ 0.137
minParams	▲ 0.036
maxParams	⊙ 0.178
avgParams	⊙ 0.074
totalParams	⊙ 0.437
Time (ms)	⊙ 0,966

this is due to the way these algorithms explore the search space. Fig. 4(b) compares the final SSCC of the original method after the cognitive complexity reduction. This boxplot confirms that ES-LSF tends to reduce SSCC more than ES-SSF. In fact, differences are significant between these two algorithms for this metric. Figs. 4(c), 4(d), and 4(e) can be interpreted all together as the behaviour of the presented approaches slightly affect them in the same way. The results confirm our expectations again. In fact, differences are significant between the two algorithms for `avgReduc` and `avgLOC` metrics. Finally, Fig. 4(f) shows the execution time in seconds. Both algorithms took almost 20 hours to process all methods with cognitive complexity issues on the 10 software projects under study. Despite the execution time looks similar, there are some outlier solutions in the case of ES-SSF that take longer time to meet the stopping criteria. The outlier near 800 seconds of execution time represent unique method which ES-SSF is able to obtain a solution but ES-LSF is not. Both algorithms took, on average, less than 70 seconds to reduce methods cognitive complexity.

In order to analyze the overall performance of the proposed approach to automatize the cognitive complexity reduction task, Table 5 shows aggregated metrics (min, max, mean, and standard deviation) for each project.

As shown, all projects, excepting Fiware-Commons, require, on average, more than one code extraction to reduce the SSCC of their methods to 15. However, five projects (CyberCaptor, FastJson, Jmetal, Knowage-core, and MOEA) required five or more code extractions to reduce the SSCC of some methods. In general, code extractions reduced, on average, SSCC by 12 units. Nevertheless, some code extractions reduced methods SSCC up to 72 units.

1) ANALYZING NOT SO TYPICAL SOLUTIONS

Next, we explain not so typical solutions obtained in this experiment which can be detected analyzing the results shown in Tables 3 and 5.

- Short code extractions. There are 10 solutions where an extraction of only one line of code is enough to reduce the SSCC of the method. A extraction of this kind is the following:

```
Boolean b = Boolean.valueOf(str ? true : false)
```

The ternary operator is a part of Java's conditional statements. As the name ternary suggests, it is the only operator in Java consisting of three operands. The ternary operator can be thought of as a simplified version of the `if-else` statement with a value to be returned. This kind of statement contributes to the SSCC of a method with its *inherent component* ($\iota = 1$) plus the *nesting component*, which depends on the nesting level of the statement in the source code. Therefore, when extracting this kind of one-line statement the cognitive complexity of a method might be reduced by at least one.

- Too many code extractions. There is a solution provided by the ES-SSF that suggests 30 extractions. This happens in the `decodeDispatchContext` method in the `SchedulerUtilities` class of the Knowage-core project. This method has 154 lines of code and its SSCC is 86. It has 37 `if` statements at the same level of nesting. This results in too many extractions when applying the ES-SSF algorithm. Note that increasing the number of evaluations, the number of extraction

TABLE 5. Statistics for the cognitive complexity reduction task performed over the 10 projects under study.

Project	Metric	iniCC	extrac	reducCC	finalCC
ByteCode	Mean	26.35	1.33	13.98	12.37
	Std. Deviation	11.536	0.659	12.195	2.919
	Minimum	16	1	1	3
	Maximum	63	4	54	15
CyberCaptor	Mean	24.36	1.39	13.62	10.73
	Std. Deviation	10.023	1.021	11.602	3.975
	Minimum	16	1	1	1
	Maximum	52	5	48	15
FastJson	Mean	25.43	1.47	13.37	12.06
	Std. Deviation	9.895	0.828	9.497	3.794
	Minimum	16	1	2	1
	Maximum	76	5	62	15
Fiware-Commons	Mean	17	1	5	12
	Std. Deviation	1.549	0	0	1.549
	Minimum	16	1	5	11
	Maximum	19	1	5	14
IoTBroker	Mean	19.56	1.11	7.83	11.72
	Std. Deviation	6.845	0.323	6.653	2.296
	Minimum	16	1	3	6
	Maximum	38	2	24	15
Jedis	Mean	25.5	1.5	11.5	14
	Std. Deviation	2.887	0.577	1.732	1.155
	Minimum	23	1	10	13
	Maximum	28	2	13	15
Jmetal	Mean	26.2	1.39	13.35	12.85
	Std. Deviation	10.285	0.874	10.388	3.493
	Minimum	16	1	1	1
	Maximum	56	6	44	15
Knowage-core	Mean	26.22	1.47	13.88	12.33
	Std. Deviation	10.582	1.324	10.942	3.364
	Minimum	16	1	1	1
	Maximum	86	30	72	15
MOEA	Mean	23.55	1.3	11.36	12.19
	Std. Deviation	8.542	0.964	9.753	3.822
	Minimum	16	1	1	1
	Maximum	53	11	52	15
QueryExecution	Mean	25.67	1.33	13.67	12
	Std. Deviation	10.316	0.492	11.934	3.516
	Minimum	17	1	5	5
	Maximum	41	2	33	15

```
1  //Cognitive complexity 8
2  public String addParametersToServiceUrl(...) throws
       ...
3  {
4   List<BIObjectParameter> drivers = ...;
5   // +1
6   if (drivers != null) {
7    List<Parameter> parameter = ...;
8    // +2 (1 by nesting)
9    if (drivers.size() != parameter.size()) {
10     throw new SpagoBIRuntimeException("There are...");
11   }
12   Collections.sort(drivers);
13   ParametersDecoder decoder = new ParametersDecoder();
14   // +2 (1 by nesting)
15   for (BIObjectParameter biObjectParameter : drivers)
        {
16    boolean found = extraction1(...); // 5 parameters
17    // +3 (2 by nesting)
18    if (!found) {
19     throw new SpagoBIRuntimeException("...");
20    }
21   }
22  }
23  return serviceUrlBuilder.toString();
24 }
```

FIGURE 5. Method `addParametersToServiceUrl` of the open-source project knowage-core after reducing its SSCC from 46 to 8. Three Extract Method refactoring operations were needed.

decreases. For example, using 100,000 and one million evaluations, ES-SSF performs 28 and 25 extractions, respectively.

- Many parameters in extracted methods. There is a solution that extracts a new method with 15 parameters. This happens in the `manageRecursiveSection` method in the `HierarchyMasterService` class of the Knowage-core project. The reason is that the original method has 16 parameters. The best solution for this case is a single extraction of 32 lines of code which reduces the SSCC of the original method in 19. However, most original parameters are needed in the extracted method.

2) COMING BACK TO THE RUNNING EXAMPLE

Following with the running example introduced in Section III, Fig. 5 shows the `addParametersToServiceUrl` method after cognitive complexity reduction. The SSCC of this method has been reduced from 46 to 8 after applying three Extract Method refactoring operations. The number of lines of code has also reduced from 65 to 24. Note that, although three Extract Method operations are applied, just one appears in the code (line 16). The reason is that the other two method extractions are called from the extracted method `extraction1`. These two additional method extractions are required to reduce the SSCC of the first extracted method to τ.

VII. DISCUSSION

A number of cognitive complexity metrics have been proposed in the literature measuring software cognitive complexity in different ways. However, software complexity has gained popularity last years due to the usage of SonarCloud and SonarQube as service and platform, respectively, for continuous inspection of code quality. For this reason we used the cognitive complexity measure provided by these well-known static code tools in this work, which we referred to SSCC. In order to search for feasible refactoring opportunities to reduce cognitive complexity of a method, the proposed approach generates its Abstract Syntax Tree (AST) and annotates in their nodes information about the contribution to the cognitive complexity of the method. Thus, other cognitive complexity measures which take into account the presence of control flow structures in source code for their computation could be integrated in our tool: it would just require to adapt the way the approach gets the list of existing cognitive complexity issues in a project and the computation of the properties annotated in the nodes of the AST of the methods. However, this is out of the scope of the paper: our main contribution is that software cognitive complexity reduction can be modeled as an optimization problem and automatizing SSCC reduction is feasible, which is empirically proven through our experiments.

Concerning the resolution process, a block of consecutive statements can be extracted if pre-conditions and post-conditions are met and the extraction generates compilable code. The more blocks of consecutive statements that are extractable, the more possibilities the cognitive complexity can be reduced. However, developers do not know which sequences of statements are extractable but also the impact of any code extraction on code cognitive complexity. Therefore, developers cannot make informed decisions concerning the cognitive complexity of a method when maintaining its code. The resolution techniques proposed here have a number of advantages. Among them, it stands out that they achieve optimal solutions quickly for most of the methods analyzed (78%) without using any heuristic or randomized operator. In contrast, they require a high number of evaluations to find feasible solutions in methods with high number of extractable blocks of code. As the number of evaluations increases, the better the solution obtained. However, the minimum number of evaluations required to find optimal solutions is unknown beforehand. If execution time is a constraint when reducing the cognitive complexity of a method, search-based software engineering techniques could be applied instead of the exhaustive algorithms used in this paper (note that our tool is algorithm independent when solving the cognitive complexity reduction problem). Nevertheless, the proposed approach and the implemented resolution techniques took, on average, less than 70 seconds to process each method of the 10 software projects in our case study. It seems reasonable to integrate software cognitive complexity reduction in the development workflow (e.g., using continuous integration). Thus, the proposed approach could be automatically run at night or after developers commit new changes to software repositories.

In this work we decided to minimize the number of Extract Method refactoring operations when reducing the SSCC of a method. Therefore, all applicable sequences of extractions with minimum size are optimal. However, not all these sequences have the same characteristics and they vary in most metrics studied in this paper: extracted lines of code, extracted SSCC, number of parameters, final SSCC in the original method, and many others. It is possible to

model the cognitive complexity reduction problem as a multi-objective or many-objective optimization problem. In that case different techniques can be applied to optimize several of these metrics at the same time. In addition, multiple criteria decision making could be applied allowing developers to decide which solutions seem most appropriate for them based on their preferences.

Interestingly, we found that some coding practices could hinder the cognitive complexity reduction task. This usually happens when methods contain multiple `return` statements. Having too many `return` statements in a method decreases the method's essential understandability because the flow of execution is broken each time a `return` statement is encountered. This makes it harder to read and understand the logic of the method but could also prevent the extraction of the code.[10] Consequently, the use of multiple `return` statements in a method might make an instance of the cognitive complexity reduction problem unsolvable. This also holds for loops containing multiple `break` or `continue` statements, which also breaks the execution flow. Therefore, restricting the number of `break` and `continue` statements in a loop is done in the interest of good structured programming.[11]

> *The use of multiple* `return`*,* `break`*, and* `continue` *statements in a method could hinder the cognitive complexity reduction task or even make it unsolvable.*

An aspect that is out of the scope of this article is the choice of the name for the new extracted methods. The name of new methods can influence the understanding of the resulting source code. Therefore, this is an important aspect we plan to address in the near future. Creating a dictionary with keywords in the original method and using natural language processing techniques with Transformers [37] could be a good starting point to handle this fact.

VIII. THREATS TO VALIDITY

This section discusses all threats that might have an impact on the validity of our study following common guidelines for empirical studies [38].

Threats to internal validity concern factors that could have influenced our results. A possible threat to internal validity is that we set a stopping criteria of 10,000 evaluations. This stop condition might have influenced our results because the algorithms, in some cases, stop before all possible sequences of extractions are explored. However, in order to alleviate this issue, we have presented two completely different ways of exploring the search space. Another aspect that can influence the results is the choice of the cognitive complexity metric and the used threshold. A number of cognitive complexity measures have been proposed in the literature. However, there is no single metric which has the capability of measuring the complexity of a program based on multiple object-oriented concepts [6]. We used the cognitive complexity metric integrated in the well-known static code tools SonarCloud and SonarQube because (i) this metric positively correlates with source code understandability [9] and has a 77% acceptance rate among developers [8], (ii) it is accessible via SonarQube API REST, and (iii) a well-defined cognitive complexity threshold is suggested for it.

Threats to construct validity concern relationship between theory and observation and the extent to which the measures represent real values. In our study all the experiments were run in the same computer and the metrics we collected are all consistent when analyzing the original and the resulting source codes.

Threats to external validity concern the generalization of our findings. To reduce external validity threats we selected a diverse set of 10 open-source projects for our case of study. Aggregating all projects, we processed 1,050 methods with SSCC greater than 15. This high number of existing issues guarantees that we have analyzed very diverse methods in terms of complexity and size. Thus, we guess our findings can be generalized to other software projects.

Threats to conclusion validity concern the relationship between experimentation and outcome. We compared the results of two different variants of an exhaustive algorithm and performed a Mann-Whitney-Wilcoxon test to determine the statistical significance of the results. In addition, a large number of methods were analyzed and the algorithms had enough evaluations to find feasible solutions for most of the methods of the software projects under study.

IX. CONCLUSION

We formulated the reduction of software cognitive complexity provided by SonarCloud and SonarQube, to a given threshold, as an optimization problem. We then proposed an approach to automatically reduce the cognitive complexity of methods in software projects to the chosen threshold using sequences of Extract Method refactorings. We conducted some experiments in 10 open-source software projects analyzing more than 1,000 methods with a cognitive complexity greater than the default threshold suggested by SonarQube (15). Our automated approach was able to reduce the cognitive complexity to or below the threshold in 78% of those methods.

We found that statements that brake the execution flow of programs could prevent the extraction of code and, therefore, make a particular instance of the cognitive complexity reduction problem unsolvable. With the aim of helping to alleviate this issue, we propose as future work a semantically equivalent code transformation that increases the number of extractable blocks of code in a method by reducing the number of `return`, `break`, and `continue` statements. This transformation will indirectly improve the readability and maintainability of the code, but it will also benefit the cognitive complexity reduction task.

Although our approach was able to reduce the cognitive complexity for most methods, we cannot assure that no solution exist for the rest of the methods. The reason is that the cost of exploring all possible sequences of extractions might

[10] https://rules.sonarsource.com/java/RSPEC-1142?search=return
[11] https://rules.sonarsource.com/java/RSPEC-135?search=break

be unaffordable. However, we think that it is preferable to maintain the simplicity of the approach to emphasize the benefits of providing an automated tool. As future work we want to study the NP-hardness of the modeled cognitive complexity reduction problem. If we prove this, a different procedure (like an ad-hoc metaheuristic) could be included in our approach to solve the problem. Last but not least, we plan to validate our approach on software developers in order to get their feedback and analyze the way of including our approach as part of the continuous integration practice.

APPENDIX

In the field of theoretical validation, a number of researchers have proposed different criteria to which software measures should adhere. Weyuker established a formal list of nine properties in order to estimate the accuracy of software metrics [39]. It has been used to evaluate numerous existing software metrics. Next we evaluate the cognitive complexity metric provided by SonarCloud and SonarQube (which we refer to as SSCC) against Weyuker's properties and validate it against measurement theory, as suggested in the framework proposed by Misra *et al.* [28]. Then, we perform a practical validation with Kaner's framework [40]. We finally end up with a comparative analysis and conclusion of the validation.

A. WEYUKER'S PROPERTIES

In the following, P, Q, and R are methods of a class. With $|P|$ and $(P; Q)$ we refer to the SSCC of method P and the composition of P and Q methods, respectively.

1) PROPERTY 1. $(\exists P)(\exists Q)(|P| \neq |Q|)$, WHERE P AND Q ARE TWO DISPARATE METHODS

By definition of the measure, usually different methods have different SSCC values. Hence, this property holds for this measure.

2) PROPERTY 2: LET c BE A NON-NEGATIVE NUMBER THEN THERE ARE ONLY FINITELY MANY METHODS OF COMPLEXITY c

All projects have finite number of classes and methods, and all methods have a finite number of statements. Because SSCC of a method depends on its statements then there are only finitely many methods that will be equal to the measure c. The SSCC metric thus holds for this property.

3) PROPERTY 3: THERE ARE DISTINCT METHODS P AND Q SUCH THAT $|P| = |Q|$

This property says that there can be multiple methods containing the same SSCC value. Two methods without control flow structures will have equal complexity (0). Hence this property is satisfied by this measure.

4) PROPERTY 4: $(\exists P)(\exists Q)(P \equiv Q$ AND $|P| \neq |Q|)$

This property states that even though two methods compute the same function, it is the details of the implementation that determine the methods complexity. Even though the functionalities of two methods are equal, their complexity depends on the number of control flow structures and their nesting level on the code. Because of that the SSCC measure holds this property.

5) PROPERTY 5: $(\forall P)(\forall Q)(|P| \leq |P; Q|$ AND $|Q| \leq |P; Q|)$

This property states that the complexity values of two methods should be less than or equal to the complexity of the composition of the two methods. The SSCC measure mainly depends on the presence of control flow structures and complex expressions which determine the *inherent component* complexity of a method. Thus, the complexity value of the combination of two methods should be greater than or equal to the complexity value of these two methods. Hence this property is satisfied by this measure.

6) PROPERTY 6: $(\exists P)(\exists Q)(\exists R)(|P| = |Q|)$ AND $(|P; R| \neq |Q; R|)$

This property states that if there are two methods P and Q with same SSCC and when they are separately combined with the same third method R, yields a method of different SSCC. For any two methods P and Q, any combination of them with another method R will produce new methods with similar SSCC. Therefore, this measure does not satisfy this property.

7) PROPERTY 7: THERE ARE METHODS P AND Q SUCH THAT Q IS FORMED BY PERMUTING THE ORDER OF THE STATEMENTS OF P AND $(|P| \neq |Q|)$

Changing the order of the statements in a method, without changing the functionality of the method, will not change its complexity value. Therefore, this measure does not satisfy this property.

8) PROPERTY 8: IF P IS RENAMING OF Q, THEN $|P| = |Q|$

Renaming of a method does not impact its SSCC. As a consequence, this property is satisfied by this measure.

9) PROPERTY 9: $(\exists P)(\exists Q)(|P| + |Q| < |P; Q|)$

This property states that the addition of complexities of two separate methods is lower than the complexity of a method which is created by joining those two separate methods. The SSCC of the combined method never reduces. Because of this situation this condition is not fulfilled by this metric. However, the modified version of this property, $(\exists P)(\exists Q)(|P| + |Q| \leq |P; Q|)$ [33], is satisfied.

Table 6 gives a summary of the evaluation process through Weyuker's properties. The satisfied properties are marked. As shown, and according to the above explanation, the SSCC measure satisfies all the properties of Weyuker's framework excluding 7th and 9th. Thus, we suggest that the SSCC measure establishes as a well-structured one.

B. MEASUREMENT THEORY

Here we validate the SSCC measure against measurement theory using the Briand *et al.* framework [41]. We do our

TABLE 6. Summary of evaluation of the SSCC metric through Weyuker's properties.

Property	1	2	3	4	5	6	7	8	9
Satisfied	✓	✓	✓	✓	✓	X	X	✓	✓

assessment providing the basic definitions and desirable properties that make up the framework.

Definition (Representation of Systems and Modules): "A system S is represented as a pair $< E, R >$, where E represents the set of elements of S, and R is a binary relation on E ($R \subseteq E \times E$) representing the relationships between S's elements.".

For the SSCC metric, E can be defined as a method containing a set of code statements and R as the set of *inherent components*: complex expressions and control flows from one statement to another.

Definition (Complexity): "The complexity of a system S is a function Complexity (S) that is characterized by the following five properties: non-negativity, null value, symmetry, module monotonicity, and disjoint module additive.".

- Non-negative: "*The complexity of a system $S =< E, R >$ is non-negative if Complexity(S) ≥ 0.*".
  SSCC values are always positive, this property is thus satisfied by this measure.
- Null value: "*The complexity of a system $S =< E, R >$ is null if R is empty.*".
  If a method does not contain *inherent components* (certain control flow structures or complex expressions), then it will have null (0) complexity. Thus, this property is satisfied by the SSCC metric.
- Symmetry: "*The complexity of a system $S =< E, R >$ does not depend on the convention chosen to represent the relationships between its elements.*".
  There is no effect on the complexity values of the SSCC metric by changing the order or representation because the contribution of control flow structures and its nesting level to the overall complexity of a method cannot depend on the order or way of representation. Therefore, this property is satisfied by the SSCC measure.
- Module monotonicity: "*The complexity of a system $S =< E, R >$ is no less than the sum of the complexities of any two of its modules with no relationships in common.*".
  For the SSCC metric, a module can be defined as a code segment in a method. For this property, if any method is partitioned into two methods, the sum of the complexity values of its partitioned methods will never be greater than the one of the joined method. Therefore, this property holds for the metric.
- Disjoint Module Additivity: "*The complexity of a system $S =< E, R >$ composed of two disjoint modules m_1 and m_2 is equal to the sum of the complexities of the two modules.*".
  The SSCC value of the method obtained by concatenating m_1 and m_2 is equal to the sum of their calculated complexity values. Thus, if two independent methods are combined into a single method then the complexity of the individual methods will be combined. Therefore, this property is satisfied by the SSCC measure.

By fulfilling these properties, one may say that the SSCC metric is on the ratio scale, which is the most desirable property of complexity measures from the point of view of measurement theory [33].

C. PRACTICAL VALIDATION WITH KANER'S FRAMEWORK

In addition to the theoretical validation using Weyukers' properties and measurement theory, the framework given by Kaner [40] can also be adopted for evaluation of the SSCC metric. This approach is more practical than the formal approach of Weyukers' properties and measurement theory. The framework is based on providing answers to the following points:

1) PURPOSE OF THE MEASURE

The purpose of the measure is to evaluate the complexity of methods in object-oriented programming languages.

2) SCOPE OF THE MEASURE

Object-orientation is widely adopted nowadays in the development of software, from open-source to proprietary software. The SSCC measure can be used within and across these projects.

3) IDENTIFIED ATTRIBUTE TO MEASURE

The SSCC metric is defined as a measure of how hard the control flow of a method is to understand and maintain. Thus, The identified attributes that the SSCC measure addresses are understandability and maintainability.

4) NATURAL SCALE OF THE ATTRIBUTE

The natural scales of the attributes cannot be defined, since it is subjective and requires the development of a common view about them [33].

5) NATURAL VARIABILITY OF THE ATTRIBUTE

Natural variability of the attributes can also not be defined because of their subjective nature. It is possible that one can develop a sound approach to handle such attribute, but it may not be complete because other factors also exist that can affect the attribute's variability [33].

6) DEFINITION OF METRIC

The SSCC metric has been formally defined by Campbell [42] and briefly introduced in Section I.

7) MEASURING INSTRUMENT TO PERFORM THE MEASUREMENT

The SSCC measure was computed by SonarQube.

8) NATURAL SCALE FOR THE METRIC

The SSCC measure is on the ratio scale, as mentioned earlier in this section.

9) RELATIONSHIP BETWEEN THE ATTRIBUTE AND THE METRIC VALUE

The SSCC metric contributes to determining the overall complexity of methods and classes in object-oriented programming languages. Higher SSCC values translate into code harder to understand and maintain.

10) NATURAL FORESEEABLE SIDE EFFECTS OF USING THE INSTRUMENT

There are no side effects of using SonarQube to measure the SSCC of software projects because the computation of the metric is automatically performed by it. In addition to the previous, SonarQube is an open-source tool and its source code is available in public repositories.

D. COMPARATIVE ANALYSIS AND CONCLUSION OF THEORETICAL VALIDATION

The SSCC measure satisfied seven out of the nine Weyuker's properties. Although these results were convincing enough, we turned to the measurement theory. Measurement theory has five properties all of which were satisfied by the SSCC metric. This shown that this measure is additive and on the ratio scale. Finally, the Kaner's framework was used to prove the usefulness of the SSCC measure after asking practical questions.

REFERENCES

[1] D. Coleman, D. Ash, B. Lowther, and P. Oman, ''Using metrics to evaluate software system maintainability,'' *Computer*, vol. 27, no. 8, pp. 44–49, Aug. 1994.

[2] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Hoboken, NJ, USA: Wiley, 2011.

[3] UNDO. (2017). *Increasing Software Development Productivity With Reversible Debugging*. [Online]. Available: https://undo.io/media/uploads/files/Undo_ReversibleDebugging_Whitepap%er.pdf

[4] D. S. Kushwaha and A. K. Misra, ''Cognitive complexity metrics and its impact on software reliability based on cognitive software development model,'' *ACM SIGSOFT Softw. Eng. Notes*, vol. 31, no. 2, pp. 1–6, Mar. 2006, doi: 10.1145/1118537.1118544.

[5] S. T. Rabani and K. Maheswaran, ''Software cognitive complexity metrics for OO design: A survey,'' *Int. J. Sci. Res. Sci., Eng. Technol.*, vol. 3, pp. 692–698, 2017.

[6] T. Jayalath and S. Thelijjagoda, ''A modified cognitive complexity metric to improve the readability of object-oriented software,'' in *Proc. Int. Res. Conf. Smart Comput. Syst. Eng. (SCSE)*, Sep. 2020, pp. 37–44.

[7] T. McCabe, ''A complexity measure,'' *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[8] G. A. Campbell, ''Cognitive complexity: An overview and evaluation,'' in *Proc. Int. Conf. Tech. Debt*, May 2018, pp. 57–58, doi: 10.1145/3194164.3194186.

[9] M. Muñoz Barón, M. Wyrich, and S. Wagner, ''An empirical validation of cognitive complexity as a measure of source code understandability,'' in *Proc. 14th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Oct. 2020, pp. 1–12, doi: 10.1145/3382494.3410636.

[10] D. Silva, N. Tsantalis, and M. T. Valente, ''Why we refactor? Confessions of GitHub contributors,'' in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2016, pp. 858–870, doi: 10.1145/2950290.2950305.

[11] N. Tsantalis and A. Chatzigeorgiou, ''Identification of extract method refactoring opportunities for the decomposition of methods,'' *J. Syst. Softw.*, vol. 84, no. 10, pp. 1757–1782, Oct. 2011, doi: 10.1016/j.jss.2011.05.016.

[12] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, ''A multidimensional empirical study on refactoring activity,'' in *Proc. Conf. Center Adv. Stud. Collaborative Res. (CASCON)*. Armonk, NY, USA: IBM Corp., 2013, pp. 132–146.

[13] A. Hora and R. Robbes, ''Characteristics of method extractions in java: A large scale empirical study,'' *Empirical Softw. Eng.*, vol. 25, no. 3, pp. 1798–1833, May 2020, doi: 10.1007/s10664-020-09809-8.

[14] R. Haas and B. Hummel, ''Deriving extract method refactoring suggestions for long methods,'' in *Software Quality. The Future of Systemsand Software Development*, D. Winkler, S. Biffl, and J. Bergsmann, Eds. Cham, Switzerland: Springer, 2016, pp. 144–155.

[15] J. Hubert, ''Implementation of an automatic extract method refactoring,'' M.S. thesis, Fac. Comput. Sci., Elect. Eng. Inf. Technol., Univ. Stuttgart, Stuttgart, Germany, Germany, Apr. 2019.

[16] M. H. Halstead, *Elements of Software Science*, M. H. Halstead, Ed. New York, NY, USA: Elsevier, 1977.

[17] M. R. Woodward, M. A. Hennell, and D. Hedley, ''A measure of control flow complexity in program text,'' *IEEE Trans. Softw. Eng.*, vol. SE-5, no. 1, pp. 45–50, Jan. 1979.

[18] C. R. Douce, P. J. Layzell, and J. Buckley, ''Spatial measures of software complexity,'' in *Proc. PPIG*. Delft, The Netherlands: Psychology of Programming Interest Group, 1999, p. 6.

[19] J. Shao and Y. Wang, ''A new measure of software complexity based on cognitive weights,'' in *Proc. CCECE Can. Conf. Electr. Comput. Eng. Toward Caring Humane Technol.*, vol. 2, May 2003, pp. 1333–1338.

[20] S. Misra, ''Modified cognitive complexity measure,'' in *Computer and Information Sciences ISCIS 2006*, A. Levi, E. Savaş, H. Yenigün, S. Balcısoy, and Y. Saygın, Eds. Berlin, Germany: Springer, 2006, pp. 1050–1059.

[21] S. Misra, ''A complexity measure based on cognitive weights,'' *Int. J. Theor. Appl. Comput. Sci.*, vol. 1, no. 1, pp. 1–10, 2006.

[22] S. Misra and A. K. Misra, ''Evaluation and comparison of cognitive complexity measure,'' *ACM SIGSOFT Softw. Eng. Notes*, vol. 32, no. 2, pp. 1–5, Mar. 2007, doi: 10.1145/1234741.1234761.

[23] S. Misra, ''Cognitive program complexity measure,'' in *Proc. 6th IEEE Int. Conf. Cognit. Informat.*, Aug. 2007, pp. 120–125.

[24] S. Misra, ''An object oriented complexity metric based on cognitive weights,'' in *Proc. 6th IEEE Int. Conf. Cognit. Informat.*, Aug. 2007, pp. 134–139.

[25] S. Misra and I. Akman, ''A model for measuring cognitive complexity of software,'' in *Knowledge-Based Intelligent Information and Engineering Systems*, I. Lovrek, R. J. Howlett, and L. C. Jain, Eds. Berlin, Germany: Springer, 2008, pp. 879–886.

[26] S. Misra and I. Akman, ''A new complexity metric based on cognitive informatics,'' in *Rough Sets and Knowledge Technology*, G. Wang, T. Li, J. W. Grzymala-Busse, D. Miao, A. Skowron, and Y. Yao, Eds. Berlin, Germany: Springer, 2008, pp. 620–627.

[27] S. Misra, I. Akman, and M. Koyuncu, ''An inheritance complexity metric for object-oriented code: A cognitive approach,'' *Sadhana*, vol. 36, no. 3, p. 317, Jul. 2011, doi: 10.1007/s12046-011-0028-2.

[28] S. Misra, M. Koyuncu, M. Crasso, C. Mateos, and A. Zunino, ''A suite of cognitive complexity metrics,'' in *Computational Science and its Applications ICCSA 2012*, B. Murgante, O. Gervasi, S. Misra, N. Nedjah, A. M. A. C. Rocha, D. Taniar, and B. O. Apduhan, Eds. Berlin, Germany: Springer, 2012, pp. 234–247.

[29] S. Misra, I. Akman, and R. Colomo-Palacios, ''Framework for evaluation and validation of software complexity measures,'' *IET Softw.*, vol. 6, no. 4, pp. 323–334, Aug. 2012. [Online]. Available: https://digital-library.theiet.org/content/journals/10.1049/iet-sen.201%1.0206

[30] D. R. Wijendra and K. P. Hewagamage, ''Automated tool for the calculation of cognitive complexity of a software,'' in *Proc. 2nd Int. Conf. Sci. Inf. Technol. (ICSITech)*, Oct. 2016, pp. 163–168.

[31] M. Crasso, C. Mateos, A. Zunino, S. Misra, and P. Polvorín, ''Assessing cognitive complexity in java-based object-oriented systems: Metrics and tool support,'' *Comput. Inform.*, vol. 35, no. 3, pp. 497–527, 2016. [Online]. Available: http://www.cai.sk/ojs/index.php/cai/article/view/1747

[32] S. Misra, A. Adewumi, R. Damasevicius, and R. Maskeliunas, ''Analysis of existing software cognitive complexity measures,'' *Int. J. Secure Softw. Eng.*, vol. 8, no. 4, pp. 51–71, Oct. 2017, doi: 10.4018/IJSSE.2017100103.

[33] S. Misra, A. Adewumi, L. Fernandez-Sanz, and R. Damasevicius, ''A suite of object oriented cognitive complexity metrics,'' *IEEE Access*, vol. 6, pp. 8782–8796, 2018.

[34] J. C. Cherniavsky and C. H. Smith, ''On Weyuker's axioms for software complexity measures,'' *IEEE Trans. Softw. Eng.*, vol. 17, no. 6, pp. 636–638, Jun. 1991, doi: 10.1109/32.87287.

[35] L. Kaur and A. Mishra, "Cognitive complexity as a quantifier of version to version java-based source code change: An empirical probe," *Inf. Softw. Technol.*, vol. 106, pp. 31–48, Feb. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584918301903
[36] B. S. Alqadi, "The relationship between cognitive complexity and the probability of defects," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2019, pp. 600–604.
[37] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, U. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst. (NIPS)*. Red Hook, NY, USA: Curran Associates Inc., 2017, pp. 6000–6010.
[38] R. K. Yin, *Case Study Research: Design Methods*, 3rd ed. Newbury Park, CA, USA: Sage, 2002.
[39] E. J. Weyuker, "Evaluating software complexity measures," *IEEE Trans. Softw. Eng.*, vol. SE-14, no. 9, pp. 1357–1365, Sep. 1988. [Online]. Available: http://ieeexplore.ieee.org/document/6178/
[40] C. Kaner and W. P. Bond, "Software engineering metrics: What do they measure and how do we know," in *Proc. Int. Softw. Metrics Symp.*, Washington, DC, USA: IEEE Computer Society Press, 2004, pp. 1–12.
[41] L. C. Briand, S. Morasca, and V. R. Basili, "Property-based software engineering measurement," *IEEE Trans. Softw. Eng.*, vol. 22, no. 1, pp. 68–86, Jan. 1996.
[42] G. A. Campbell, "Cognitive complexity—A new way of measuring understandability," SonarSource, Geneva, Switzerland, White Paper, 2017. [Online]. Available: https://www.sonarsource.com/docs/CognitiveComplexity.pdf

RUBÉN SABORIDO received the B.S. degree in computer engineering and the M.S. degree in software engineering and artificial intelligence from the University of Málaga, Spain, and the Ph.D. degree in computer engineering from Polytechnique Montréal, Canada, in 2017. He worked three years as a Researcher Assistant at the University of Málaga. He is currently a Researcher at the Networking and Emerging Optimization Group, Department of Computer Science, University of Málaga. In 2018, he held a postdoctoral fellowship at Concordia University, Canada, where he worked on search-based software engineering for the Internet of Things (IoT). He was awarded a Juan de la Cierva Grant, in 2020, funded by the Spanish State Research Agency, and ranked in the top five of the candidates in the information and communications technologies area. His research interest includes search-based software engineering. He is also interested in the use of metaheuristics to solve multidisciplinary real-world problems of interest for our society and computer science. He has published several papers in ISI indexed journals (such as *EMSE*, IEEE Transactions on Software Engineering, *Evolutionary Computation*, and *Swarm and Evolutionary Computation*) and conference papers in IEEE ICPC, MCDM, IEEE SANER, and ACM ESEC/FSE. He also acts as a reviewer of prestigious international journals (ISI/JCR). His Ph.D. thesis was nominated for best thesis award. He has co-organized the International Conference on Multiple Criteria Decision Making, in 2013. He is on the organizing committee of the 1st, 2nd, 3rd, and 4th International Workshop on Software Engineering Research and Practices for the Internet of Things (SERP4IoT), co-located with ICSE 2019, ICSE 2020, ICSE 2021, and ICSE 2022, respectively. He has been on the program committee of the Real World Applications (RWA) track of the Genetic and Evolutionary Computation Conference (GECCO), since 2016.

JAVIER FERRER received the bachelor's degree in computer science and technical computer science, the master's degree in software engineering and artificial intelligence from the University of Málaga, and the Ph.D. degree in computer engineering from the University of Málaga, in 2016.

He obtained a national FPI (Training of Research Staff) fellowship. He also holds postgraduate certificate from the University of Málaga (Pedagogical Aptitude Certificate). His research interests include optimization techniques, particularly with the use of artificial intelligence techniques applied to the fields of software engineering and, recently, of smart cities. His research works have materialized in publications in international journals (ten), book chapters (three), and international and national conference proceedings (28). Some of these works have been developed in collaboration with other research groups in countries, such as Germany, Austria, and Mexico. He has been a member of the research team in four national projects, three regional project, four project funded by the University of Málaga, and ten contracts with companies and institutions for transference of knowledge (OTRI). He also acts as a reviewer of prestigious international journals (ISI/JCR). His H-index is currently 12 with 503 cites to his works.

FRANCISCO CHICANO received the degree in physics from the National Distance Education University and the Ph.D. degree in computer science from the University of Málaga. Since 2008, he has been with the Department of Languages and Computing Sciences, University of Málaga. His research interests include the application of search techniques to software engineering problems and the use of theoretical results to efficiently solve combinatorial optimization problems. He has also been the program chair and the editor-in-chief in international events. He is in the Editorial Board of *Evolutionary Computation* journal, *Engineering Applications of Artificial Intelligence*, *Journal of Systems and Software*, *ACM Transactions on Evolutionary Learning and Optimization*, and *Mathematical Problems in Engineering*.

ENRIQUE ALBA received the degree in engineering and the Ph.D. degree in computer science from the University of Málaga, Spain, in 1992 and 1999, respectively. He works as a Full Professor with the University of Málaga with varied teaching duties, such as data communications, distributed programming, software quality, and also evolutionary algorithms, bases for R+D+i and smart cities, both at graduate and master/doctoral programs. He leads an international team of researchers in the field of complex optimization/learning with applications in smart cities, bioinformatics, software engineering, telecoms, and others. In addition to the organization of international events (ACM GECCO, IEEE IPDPS-NIDISC, IEEE MSWiM, IEEE DS-RT, and smart-CT), he has offered dozens postgraduate courses, more than 70 seminars in international institutions, and has directed many research projects (nine with national funds, seven in Europe, and numerous bilateral actions). Also, he has directed 12 projects for innovation in companies (OPTIMI, Tartessos, ACERINOX, ARELANCE, TUO, INDRA, AOP, VATIA, EMERGIA, SECMOTIC, ArcelorMittal, ACTECO, CETEM, and EUROSOTERRADOS) and has worked as an invited Professor at INRIA, Luxembourg, Ostrava, Scotland, Japan, Argentina, Cuba, Uruguay, and Mexico. He is an editor in several international journals and book series of Springer-Verlag and Wiley, as well as he often reviews articles for more than 30 impact journals. He is included in the list of most prolific DBLP authors, and has published 130 articles in journals indexed by ISI, 11 books, and hundreds of communications to scientific conferences. He is included in the top five most relevant researchers in informatics in Spain (according to ISI), and is the most influent researcher of UMA in engineering (webometrics), with 14 awards to his professional activities. His H-index is 62, with more than 18,000 cites to his work.