Annales Univ. Sci. Budapest., Sect. Comp. 57 (2024) 201–218

EVALUATION OF A RECURSION AWARE COMPLEXITY METRIC

István Bozó (Budapest, Hungary)
Zoran Budimac, Smiljana Knežev and Gordana Rakić
(Novi Sad, Serbia)
Melinda Tóth (Budapest, Hungary)

Communicated by Zoltán Horváth
(Received January 10, 2024; accepted July 2, 2024)

Abstract. Software metrics are heavily used to measure the complexity, readability, understandability, and maintainability of the source code. However, the accuracy of these metrics could be improved by considering different new aspects. In our previous research, we have introduced the Overall Path Complexity metric that takes into account recursion during the calculation of the metrics. The metric was integrated into the RefactorErl tool and evaluated for the functional programming language, Erlang. It was found that the OPC looked into new information that had not been evaluated before.

In this paper, we look into how the Overall Path Complexity metric behaves in the context of an object-oriented language, Java. During the evaluation, we used the SSQSA platform. Observed results for Java examples were compared with other complexity metrics, such as Halstead (Volume, Effort, Difficulty) Metrics, Unique Complexity Metric, Maintainability Index and Cognitive Metric. The low correlation between the OPC and other complexity metrics showed that there is new information for Java projects that have not been evaluated by the other complexity metrics so far. This research shows that besides a functional programming language, the OPC metrics bring new information for an object-oriented language too.

Key words and phrases: Software complexity, recursion, cyclomatic complexity, overall path complexity.
2010 Mathematics Subject Classification: 68N99.
Project no. TKP2021-NVA-29 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme. This work is partially supported by the CERCIRAS COST Action no. CA19135 funded by COST Association. This work is a detailed version of a MaCS 2020 presentation.

1. Introduction

Nowadays there is a high demand for easy maintenance, high-quality, and cost-efficient software products. There is a known relation between maintainability, complexity, and maintenance cost [3, 2]. By looking into different complexity attributes we can find out more about different aspects of maintainability such as easiness of coding, debugging, testing, and change proneness of software products. These attributes can be measured with complexity metrics. Complexity metrics can observe the complexity of certain parts of a software product such as functions, methods, modules, etc. or can be used to measure the overall complexity of a product. Different types of complexity metrics are designed to meet different objectives. Some elementary complexity metrics are created from the ground up and are used to derive newer types of metrics. Others focus on specific requirements and are usually language or paradigm-dependent. There are also types of metrics that aim to be general, not relying on a specific language/paradigm. Also, complexity metrics can be used to derive maintainability ones [12].

Complexity metrics evolve hand in hand with the change in how software projects are built. It is common now to use several different technologies within a single project. Also, it is not rare to mix different programming languages and paradigms within a single software solution. While some of the newer complexity metrics aim to be language/paradigm independent others focus on certain programming languages and paradigms. Some of them are built from scratch and others combine already existing metrics.

A novel approach in the field of complexity metrics is the consideration of (recursive) invocation chains on a static level by the Overall Path Complexity (OPC) [17] metric. The OPC metric belongs to the group of control-flow metrics and is built as an extension of the Cyclomatic Complexity (CC) [13]. Furthermore, the OPC is built to be language and paradigm-independent. As recursion is a universal mechanism for solving problems and is commonly used in functional programming languages, the OPC metric acknowledges the importance of evaluating it.

OPC metric has already been integrated into the RefactorErl tool [4, 21] and was used to evaluate both small examples and industrial-size projects written in Erlang programming language [17]. The results were compared with results from Halstead (Volume, Effort, Difficulty) Metrics [10], Unique Complexity Metric (UCM) [14], Maintainability Index (MI) [7] and Cognitive Metric [11]. It was observed that the correlation between the OPC and remaining complexity metrics is low. This implies that there is a new complexity aspect of software that the OPC metric takes into account.

The goal of this paper is to evaluate the OPC metrics on projects written in Java programming language and compare and contrast the results with other

complexity metrics. To achieve this similar methodology as in [17] was used. The OPC metrics along with Hastead, UCM, MI and Cognitive metrics were integrated into the SSQSA platform [15].

The rest of this paper is structured as follows. Section 2 introduces the OPC metric and reviews existing tools in which this metric is implemented. Section 3 focuses on evaluating the OPC metric on Java projects, presenting the methodology used in implementation details and the results obtained on smaller Quick Sort examples. Related work is given in Section 4, and Section 5 concludes the paper.

2. Background

Since we based our research on the OPC metric, at first, we would like to introduce it based on our previous paper [17]. Then we will introduce RefactorErl and SSQSA the static analyzers involved in our work.

2.1. OPC metric

Overall Path Complexity was introduced in [17] as an extension of Cyclomatic Complexity with added awareness of (recursive) invocations. Building blocks of OPC metrics are Cyclomatic Complexity, Inlined Cyclomatic Complexity (ICC), Length of Recursion (LOR) and Recursive Complexity (RC).

Cyclomatic Complexity is defined as the maximum number of linearly independent circuits in a Control Flow Graph (CFG) [1, 13]. If we observe a block of code with a single entry and single exit point, CC metrics [13] on a CFG graph $G = (V, E)$ can be defined as:

$$CC = e - v + 2p, \tag{2.1}$$

where e is number of edges, v number of nodes and p number of connected components.

More exit points can be allowed. We can attach exit points to entry points by adding a branch and getting another definition of CC metrics:

$$CC = e - v + p. \tag{2.2}$$

Finally, we can calculate CC metrics by counting basic predicates or nodes representing conditions with two outgoing edges:

$$CC = NoP + 1, \tag{2.3}$$

where NoP represents the number of predicates.

CFG extended with information about dependencies and communication between basic blocks and statements forms an Overall Control Flow Graph (OCFG), also known as Interprocedural (Inlined) Control Graph [20].

The algorithm used to calculate CC metrics on a CFG was applied to an OCFG to define a new metric called Interprocedural (Inlined) Complexity metrics (ICC) [17]. ICC shows the number of paths that are crossing borders by following invocations on an observed block of code.

ICC is calculated from CC by adding a CC value of each function call within the observed function. For recursive function calls, inlining is done only once for each call. Formula for calculating ICC is [17]:

$$ICC(f) = CC(f) + \sum_{g \in recCallChain} CC(g) + \sum_{g \notin recCallChain} ICC(g). \tag{2.4}$$

Iteration expressed by recursion affects the complexity more strongly than a non-recursive iteration. Also, the length of (recursive) invocation chains affects complexity growth. To measure this complexity, Length of Recursion (LOR) and Recursive Complexity (RC) were introduced.

Length of Recursion [16] represents number of nodes in a recursive chain:

$$LOR = \text{number of nodes in a recursive chain.} \tag{2.5}$$

Recursive chains are viewed as parts of a Static Call Graph (SCG) [5] that consists only of recursive functions. Single closed chains that consist of recursive functions are represented by cyclic subgraphs.

The entry point to a program and a recursive chain do not necessarily need to be fixed. A recursive chain can be entered from any node, but still, it is necessary to go through all nodes in a chain. If it is possible to enter a recursive chain through LOR nodes and for each of those nodes it's necessary to go through all LOR nodes, a metric giving all possible executions, Recursive Complexity [16] is defined as follows:

$$RC = LOR^2. \tag{2.6}$$

It is important to note that LOR and RC metrics are observed at the statement level. Every statement that is a recursive function call has its own LOR and RC value.

Overall Path Complexity is introduced in a similar way as the ICC metrics with added LOR and RC values for each recursive call. The algorithm for calculating the OPC metric is based on traversing the OCFG graph. The CC value of every entry function is calculated and for each statement within the

observed function that is a function call CC value of that function call is added. If the observed function call is not recursive, the OPC value will be the same as the ICC value. If the function call is recursive, the LOR or RC value of that specific call chain is added. Traversal ends when all branches of all chains are passed exactly once.

The paper [17] addresses two dilemmas :

1. Should OPC take into account LOR or RC?
2. Should the CC of the called function be increased or multiplied by the chosen metric?

Based on dilemmas four formulas for calculating OPC value starting from the observed function f are suggested:

$$OPC\_IL(f) = CC(f) + \sum_{g}(CC(g) + LOR(g), \tag{2.7}$$

$$OPC\_IR(f) = CC(f) + \sum_{g}(CC(g) + RC(g), \tag{2.8}$$

$$OPC\_ML(f) = CC(f) + \sum_{g}(CC(g) * LOR(g), \tag{2.9}$$

$$OPC\_MR(f) = CC(f) + \sum_{g}(CC(g) * RC(g). \tag{2.10}$$

Recursive chains that do not affect control flow, such as type-level recursion are not considered by this metric. Also, concurrency or some dynamic constructs (higher-order functions, polymorphic functions) are not the focus of these metrics.

2.2. RefactorErl

RefactorErl [4] is a static source code analyser and transformer tool for the Erlang programming language. It supports a wide variety of refactorings and code comprehension tasks. During the initial source code analysis a Semantic Program Graph (SPG) is built from the Erlang source code that contains lexical, syntactic and static semantic information about the program (such as dataflows, function call graph, variable bindings, etc.). RefactorErl provides an incremental analyser framework: once the program is changed, only the corresponding part of the SPG needs to be updated.

RefactorErl supports everyday program development tasks with different useful features. For example, it can calculate and draw dependencies among

program components, can find code duplicates, and supports gathering information about the source code with a user-level semantic query language. Several software metrics can be evaluated and queried using these semantic queries.

OPC metrics were implemented in the RefactorErl framework and all four versions of the algorithm were analysed. After analysing both general and trivial cases, OPC IR (Equation (2.8)) was found as the most appropriate for Erlang programs. Correlation analysis showed that OPC metrics introduce a new dimension of complexity which is not measured by available metrics.

2.3. SSQSA

Set of Software Quality Static Analyzers (SSQSA) is a framework with one of the main goals being to provide consistent static analysis across multiple programming languages [15]. To achieve this the framework uses unique intermediate source code representation in the form of enriched Concrete Syntax Trees (eCST). Another key feature of the SSQSA framework is language independence which is achieved by using eCST trees. This gives the framework two-level extensibility:

- support for adding new languages (adaptability),
- support for adding new analysis (extensibility).

Enriched Concrete Syntax Tree (eCST) represents a union of concepts used in abstract syntax tree (AST) and concrete syntax tree (CST). It consists of both:

- abstraction of the program structure needed for most of the program analyses,
- all concrete program elements, sometimes needed for more sophisticated program analyses.

The tool for generating eCST code representation, eCST Generator (enriched Concrete Syntax Tree Generator), is on the first level in the SSQSA framework. The resulting eCST representation is further used by other analysers in the framework. Being able to implement metrics and other analysis algorithms on eCST trees allows for easy support for new languages. Having one implementation for every analysis algorithm that is calculating metrics independently of programming language or paradigm allows for consistency, especially in heterogeneous projects.

eGDN Generator (enriched General Dependency Network Generator [19, 18]) provides a software network that represents multi-level horizontal and vertical relationships between software entities.

3. Evaluation of the OPC metric

To evaluate the OPC metrics in an object-oriented language, Java programming language has been chosen. Java is a general-purpose object-oriented programming language that is widely used. Even though iteration in Java is commonly expressed through loops, recursion can be used, too. The OPC metrics were integrated into the SSQSA platform. At the moment SSQSA supports analysis for Java 5 projects but support for newer Java versions is being developed, too.

The OPC metric was analysed on smaller Quick Sort examples implemented in Java. Most of these examples do not have long recursive invocation chains, so several quick-sort examples from [17] were rewritten in Java.

This research focused on whether the OPC metric compared to other complexity metrics will show that there is a new information, recursive invocation chains, that has not been evaluated so far that influences the complexity of the observed code. Primary data was collected by implementing CC, ICC, OPC, Halstead, Maintainability Index, UCM and Cognitive metrics on the SSQSA platform with the help of eCST and eGDN files. Data was analysed by calculating correlations between four versions of the OPC metrics with remaining complexity metrics.

SSQSA platform was chosen for its reliability in providing consistent static analysis that is language-independent. The eCST trees and eGDN network that were used contain all the necessary information at the static level for calculating all the above-mentioned complexity metrics. Furthermore, the implemented algorithms for calculating these metrics that were used on Java projects can be reused later in other languages that the platform supports.

3.1. Implementing the metrics

SSQSA platform uses an eCST generator to create specific eCST representations of input source code. The eCST representation is stored in an XML file and is further used by the eGDN generator for creating files that hold information about relationships between software entities. The output file of the eGDN generator is also stored in an XML format.

Calculating different complexity metrics was done by parsing eCST and eGDN XML files, selecting and counting specific eCST nodes and applying defined formulas for each metric.

In Figure 1 the algorithm for the calculation of OPC_IL is represented. We start by calculating the CC value of the observed function. For all function calls *apps* we use *referredFun* function to look into the ones called by an application. We further calculate their ICC values with the *icc* function and LOR values with the *contextLOR* function. Later we apply the formula (2.7) for calculating

the OPC_IL metric and return the value. By changing the formula on line 7 in this figure with formulas (2.8), (2.9), (2.10) we get algorithms for calculating OPC_IR, OPC_ML and OPC_MR.

1: $initVal \leftarrow cc(f)$
2: $apps \leftarrow functionCalls(f)$
3: **for** $a \in apps$ **do**
4: $\quad g \leftarrow referredFun(a)$
5: $\quad iccVal \leftarrow icc(g)$
6: $\quad lorVal \leftarrow contextLOR(g, f)$
7: $\quad initVal \leftarrow initVal + (iccVal + lorVal)$
8: **end for**
9: $opc \leftarrow initVal$
return opc

Figure 1. $OPC\_IL(f)$ [17]

The function *contextLOR* used for calculating LOR value in the OPC_IL algorithm is described in Figure 2. This function calculates the length of the longest recursive call chain that starts with the function f and goes through the function g. We start by looking into all call chains starting from function f. We filter them to further work with recursive call chains only. Further, we need to find such chains that start with the function f and go through function g and find the longest among them. If we have found such a chain we return its length, if not we return 0.

1: $chains \leftarrow getCallChains(f)$
2: $resChains \leftarrow filterRec(chains)$
3: $contextRec \leftarrow contextFilterLongestRec(recChains, g, f)$
4: **if** $is\_empty(contextRec)$ **then**
5: $\quad lorVal \leftarrow 0$
6: **else**
7: $\quad lorVal \leftarrow length(contextRec)$
8: **end if**
return $lorVal$

Figure 2. $contextLOR(g, f)$ [17]

The algorithm for calculating the ICC metric is described in Figure 3. We start by calculating the CC value for the observed function f. We look into all functions called by f. If there is a function called by f that is a part of a recursive call chain we add its CC value. If the called function is not part of a recursive call chain we inline it and start calculating the ICC value for it adding it to the previous result.

```
1: initVal ← cc(f)
2: apps ← functionCalls(f)
3: for a ∈ apps do
4:    g ← referredFun(a)
5:    if a ∈ recursiveChain(g, f) then
6:       initVal ← initVal + cc(g)
7:    else
8:       initVal ← initVal + icc(g)
9:    end if
10: end for
11: icc ← initVal
    return icc
```

Figure 3. $ICC(f)$ [17]

When measuring Halstead Volume, Effort and Difficulty we calculate the number of operators and operands, filter them to get the number of distinct operators and operands and apply formulas from [10] to get values for Halstead Volume, Effort and Difficulty.

The Maintainability Index is calculated with the formula:

$$MI = 171 - 5.2 * \ln(V) - 0.23 * (CC) - 16.2 * \ln(LOC) \tag{3.1}$$

Here V represents Halstead Volume, CC Cyclomatic Complexity and LOC Lines of Code.

3.2. Notes on the evaluation

Although some notions that are used in complexity metrics definitions, such as branching, loops and function calls are clearly defined, concepts like operators and operands can be viewed as ambiguous. Since there is no clear consensus on what operators and operands are, different authors and different tools use different definitions. This results in the inconsistency of results among tools [15]. The decision on what to view as operators and operands is unanimous within the SSQSA platform and all implemented complexity metrics to ensure consistent results. The choice of operators and operand definitions was influenced by choices made in the RefactorErl tool. This was done for easier comparison of gathered results from both tools, SSQSA and RefactorErl.

The decision has been made that for complexity metrics that observe function calls within the observed function, the function calls would be inlined and their metric values would be taken into consideration, too. A similar approach was used in the RefactorErl tool as well.

3.3. The used examples

Figures 4-5 presents a graphic representation of evaluated quick sort examples. From Figures 4-5 we can see that implementations of the Quick-Sort algorithm consist of two to three separate methods that call each other. In each implementation, there is at least one self-recursive call consisting of a recursive chain of length one. To illustrate that the Quick-Sort algorithm can be implemented with recursive call chains of length greater than one, two more examples with longer call chains were included.

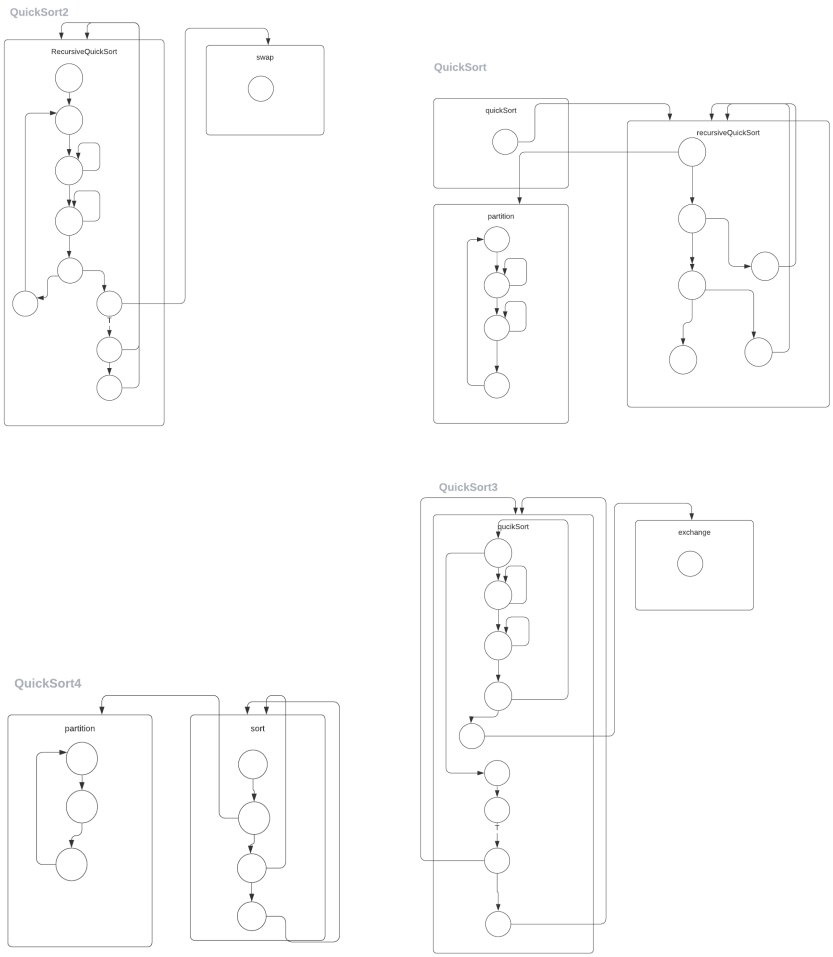

Figure 4. Graphic representation of the QuickSort implementations 1.

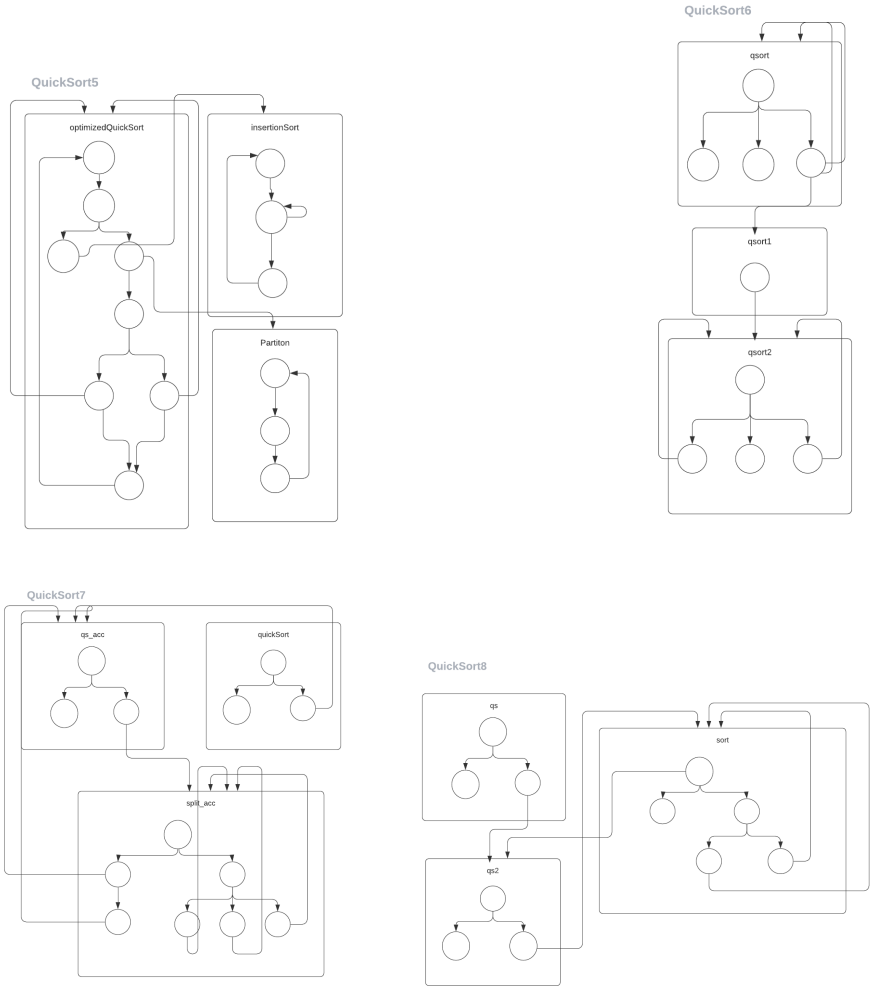

Figure 5. Graphic representation of the QuickSort implementations 2.

3.4. Evaluation

Further down we can see tables with collected *CC*, *ICC* and *OPC* data for each method of each class representing quick sort implementations.

In Table 1, CC, ICC and OPC values for each method in these *QuickSort* implementations are shown.

Class	Method	CC	ICC	OPC			
				IL	IR	ML	MR
QuickSort	recursiveQuickSort	3	14	38	38	36	36
	partition	5	5	5	5	5	5
	quickSort	1	15	15	15	15	15
QuickSort2	**recursiveQuickSort**	7	23	57	57	55	55
	Swap	1	1	1	1	1	1
QuickSort3	**quickSort**	7	22	54	54	22	22
	exchange	1	1	1	1	1	1
QuickSort4	partition	3	3	3	3	3	3
	sort	2	9	25	25	23	23
QuickSort5	insertionSort	3	3	3	3	3	3
	Partition	3	3	3	3	3	3
	optimizedQuickSort	5	21	55	55	53	53
QuickSort6	**qsort**	4	25	69	69	67	67
	qsort1	1	13	13	13	13	13
	qsort2	4	12	30	30	28	28
QuickSort7	**quickSort**	3	32	32	32	32	32
	qs_acc	3	29	41	43	75	147
	split_acc	5	36	178	182	229	345
QuickSort8	**qs**	3	24	24	24	24	24
	qs2	3	21	31	33	55	107
	sort	4	26	104	108	140	224

Table 1. Table description

Methods that represent an entry point to the *QuickSort* solution are represented in bold caption. In some cases like with methods *partition* from *QuickSort* class, *Swap* from *QuickSort2* class, etc. we can see that the CC, ICC and OPC values are all the same. This indicates that these methods do not call any other methods within their bodies, they only consist of statements such as branches and loops. In cases like in *quickSort* method from *QuickSort* class, *qsort1* from *QuickSort6 class*, etc where we can see the difference between CC values and ICC and OPC values, but ICC and OPC values are the same, we have a single method with a recursive length of one within the body of the observed method. Some methods have larger OPC_IL and OPC_IR values than OPC_ML and OPC_MR values which is because all LOR values in these methods are of length one. This is explained by the fact that OPC_IL and OPC_IR add LOR values for recursive calls, whereas OPC_ML and OPC_MR multiply values, in this case, one for the recursive calls according to formulas (2.7), (2.8), (2.9), (2.10). In *QuickSort7* and *QuickSort8* classes we can observe how the different versions of the OPC values can grow rapidly. *Split_acc* method from *QuickSort7* class has the greatest OPC values. This is because this method has

3 self-recursive calls and other recursive method calls within its body. Table 2 summarises the analysed methods and assigns an index to the algorithms. We use those indexes in Tables 3 and 4 to identify the measurements.

Index	Class	Method
1	QuckSort	quickSort
2	QuickSort2	RecursiveQuickSort
3	QuickSort3	quickSort
4	QuickSort4	quickSort
5	QuickSort5	quickSort
6	QuickSort6	qsort
7	QuickSort7	qucikSort
8	QuickSort8	qs

Table 2. Quick Sort metric values

Methods that represent an entry point to the algorithm implementations are separated in Tables 3 and 4 are further analysed with LOC, Halstead Volume, Halstead Effort, Halstead Difficulty, Unique Complexity Metric, two versions of Maintainability Index (with use of CC and with use of ICC in the formula) and Cognitive Metric. These two tables were used to calculate correlations between four versions of the OPC metrics and the other mentioned complexity metrics.

Index	CC	ICC	OPC				LOC	UCM	Cognitive
			IL	IR	ML	MR			
1	1	15	15	15	15	15	32	195	43
2	7	23	57	57	55	55	34	202	205
3	7	22	54	54	22	22	26	181	49
4	2	9	25	25	23	23	28	185	31
5	5	21	55	55	53	53	55	345	199
6	4	25	69	69	67	67	34	241	145
7	3	32	32	32	32	32	39	284	307
8	3	24	24	24	24	24	33	233	163

Table 3. Quick Sort metric values

In Table 5 correlations between four versions of the OPC metric and UCM, Halstead (Volume, Effort, Difficulty), MI and Cognitive metrics are represented. We can observe the negative correlation between the OPC metric and the Maintainability Index, which was also observed in [17] both on smaller ex-

Index	Halstead V	Halstead E	Halstead D	MI(CC)	MI(ICC)
1	518.28	15159.77	29.25	82.12	78.9
2	575.34	21685.94	37.69	79.22	75.54
3	452.36	16759.8	37.05	84.81	81.36
4	487.55	13739.97	28.18	84.37	82.76
5	973.57	49022.02	50.35	69.15	65.47
6	703.5	10567.09	15.02	78.86	74.03
7	783.6	15149.58	19.33	76.31	69.64
8	622.28	9530.66	15.32	80.9	75.38

Table 4. Quick Sort metric values

amples and bigger projects. Correlations between the OPC and other metrics vary from 0.2 to 0.4 which is similar to results in [17]. Since this is a small sample variations in results from [17] are expected.

Metric	OPC_IL	OPC_IR	OPC_ML	OPC_MR
Unique Complexity Metric	0.24	0.24	0.46	0.46
Halstead Effort	0.41	0.41	0.4	0.4
Halstead Volume	0.47	0.47	0.49	0.49
Halstead Difficulty	0.27	0.27	0.25	0.25
Maintainability Index (CC)	-0.34	-0.34	-0.59	-0.59
Maintainability Index (ICC)	-0.34	-0.32	-0.56	-0.56
Cognitive (local inlined)	0.37	0.37	0.4	0.4

Table 5. Correlation values

3.5. OPC in object-oriented projects

Some complexities can be observed at the design level in object-oriented projects, that the OPC metric does not take into account.

It is recommended to replace (large) conditionals with polymorphic methods in object-oriented projects [9]. In this way, we reduce coupling between classes and improve flexibility in the face of future changes. Even though conditionals are moved to polymorphic methods, they still exist and their complexity should be taken into account. It is left to analyse in future work how this affects the OPC metric. One possibility is to first look into the complexity of the observed method at a design level, then go down to the statement level to further calculate the complexity of a method. Polymorphic methods could be observed at a static level by looking into inheritance trees with type inference.

Another aspect to consider in future work would be object-oriented recursion. In functional languages when a function calls itself or another function recursively it happens within the same module. In object-oriented languages, we can have different instances of the same class recursively call each other. An example of that could be observed in a linked list or a tree data structure. There a structure consists of multiple instances of the same (Node) class which are connected. If we were to traverse the structure recursively we would call the same method recursively but for a different object every time. If we introduce a Null node design pattern to represent an end to a list or a tree, we would have a combination of object-oriented recursion and polymorphism, a polymorphic recursion. It should be further investigated how to enhance the OPC metric to cover these cases as well.

4. Related work

This work heavily relies on [17]. The OPC metric belongs to a group of control-flow metrics. The OPC metric aims to express the complexity of the overall problem as well as the logic that is built into it. This metric is trying to achieve this goal on a statement level while being paradigm-independent and taking the length of (recursive) invocation chains. There have been other complexity metrics that had similar objectives.

Halstead [10] looked into the complexity of the program by measuring the number of operators and operands. The difference between Halstead and the OPC metric is that Halstead calculates complexities without looking into the internal complexity of observed solutions. Effective Size Metric [6] compares executed modules to the actual number of distinct modules. Unlike the OPC metric where all possible passes are taken into account, this approach only looks into the actual number of executed passes (modules).

The most similar metrics to the OPC metric are the cognitive ones. Cognitive Functional Size [11] takes basic control structures and assigns a constant value to them. Another significant metric from the cognitive metrics group is the UCM metric [14]. The UCM metric takes into account statement-level factors that influence control flow. These factors include function calls as well as recursions. The complexity of a statement measured by several operators and operands and multiplied by a constant value is added to the overall complexity of the observed function. Improved Cognitive Based Complexity Metric [8] is defined so that the weight of a recursive call is multiplied by the weights of all factors in a recursive function. While observing recursive functions, none of these cognitive metrics look into the length of their invocation chains at the call statement.

5. Conclusion

The OPC metric is a newly developed control-flow metric whose main contribution is taking into account the length of (recursive) invocation chains at the statement/expression level. In [17] the OPC metric was defined and evaluated for the functional programming language Erlang. The OPC metric alongside Halstead (Volume, Effort, Difficulty) Metrics, Unique Complexity Metric, Maintainability Index and Cognitive Metric were integrated into the RefactorErl tool [4] and used to evaluate both small examples and bigger open source projects. The results showed that compared to other metrics, the OPC revealed new information that had not been evaluated before.

The OPC metric was designed to be paradigm-independent. In [17] the metric was examined in the context of a functional programming language. Here we evaluate the metric in the context of an object-oriented language, Java. To compare results, a similar methodology as in [17] was used. The OPC metric was integrated into the SSQSA platform. Currently, the SSQSA platform supports Java 5 projects. Java 5 examples that were evaluated were eight Quick Sort algorithm implementations. All of the examples were evaluated with the OPC metric and the results were compared with values from Halstead (Volume, Effort, Difficulty) Metric, Unique Complexity Metric, Maintainability Index and Cognitive Metric. By calculating correlations between the OPC and other mentioned metrics it was tested whether or not the OPC metric brings some new information that has not been evaluated before for Java programs.

The correlation between the OPC metric and the Maintainability Index was found to be negative. This is since in Maintainability Index the higher the value the easier it is to maintain the observed code. With other complexity metrics the situation is opposite, higher values mean the code is more difficult to maintain. Correlations with other complexity metrics varied from 0.2 to 0.4. Similar results were observed in [17]. Since this is a small sample, some deviations from results in [17] were expected. The results showed that there is new information observed on Java 5 projects that have not been evaluated by other complexity metrics. Besides functional programming languages, the OPC metric can be used for object-oriented languages too, which concludes that it is paradigm-independent.

In object-oriented languages, we can express branching through polymorphic methods. We can also observe object-oriented recursion where different instances of the same class recursively call each other. Further research needs to be done to see how the OPC metric behaves in the context of object-oriented languages. Also, evaluation of bigger Java projects needs to be done. There is work being done on supporting newer Java versions on the SSQSA platform. The SSQSA platform is language/paradigm independent future work will focus on evaluating the OPC metric for other languages supported by the platform.

References

[1] **Allen, F. E.,** Control flow analysis, in: *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, Association for Computing Machinery, 1970.

[2] **Antinyan, V., M. Staron and A. Sandberg,** Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time, *Empirical Software Engineering*, **22(6)** (2017), 3057–3087.

[3] **Banker, R. D., S. M. Datar, C. F. Kemerer and D. Zweig,** Software complexity and maintenance costs, *Commun. ACM*, **36(11)** (1993), 81–94.

[4] **Bozó, I., D. Horpácsi, Z. Horváth, R. Kitlei, J. Köszegi, M. Tejfel and M. Tóth,** RefactorErl - Source Code Analysis and Refactoring in Erlang, in: *Proceedings of the 12th Symposium on Programming Languages and Software Tools*, pages 138–148, 2011.

[5] **Callahan, D., A. Carle, M. W. Hall and K. Kennedy,** Constructing the procedure call multigraph, *IEEE Transactions on Software Engineering*, **16(4)** (1990), 483–487.

[6] **Chapin, N. and T. S. Lau,** Effective size: An example of use from legacy systems, *Journal of Software Maintenance*, **8(2)** (1996), 101–116.

[7] **Coleman, D., D. Ash, B. Lowther and P. Oman,** Using metrics to evaluate software system maintainability, *Computer*, **27(8)** (1994), 44–49.

[8] **De Silva, D. I., N. Kodagoda, S. R. Kodituwakku and A. J. Pinidiyaarachchi,** Analysis and enhancements of a cognitive based complexity measure, in: *Proceedings of IEEE International Symposium on Information Theory (ISIT)*, pages 241–245, 2017.

[9] **Ducasse, S., O. Nierstrasz and S. Demeyer,** Transform conditionals to polymorphism, in: *Proceedings of EuroPLoP'2000*, pages 219–252, 2000.

[10] **Halstead, M. H.,** *Elements of Software Science (Operating and Programming Systems Series)*, Elsevier Science Inc., 1977.

[11] **Jingqiu, S. and W. Yingxu,** A new measure of software complexity based on cognitive weights, *Canadian Journal of Electrical and Computer Engineering*, **28(2)** (2023), 69–74.

[12] **Malhotra, R. and A. Chug,** Software maintainability: Systematic literature review and current trends, *International Journal of Software Engineering and Knowledge Engineering*, **26(08)** (2016), 1221–1253.

[13] **McCabe, T. J.,** A complexity measure, *IEEE Transactions on Software Engineering*, **2(4)** (1976), 308–320.

[14] **Misra, S. and I. Akman,** A unique complexity metric, in: O. Gervasi, B. Murgante, A. Laganà, D. Taniar, Y. Mun and M. L. Gavrilova (Eds) *Computational Science and Its Applications – ICCSA 2008*, 2008, 641–651.

[15] **Rakić, G.,** SSQSA: Set of software quality static analysers, PhD thesis, Serbia, 2016.

[16] **Rakić, G., Z. Budimac, K. Bothe,** Introducing recursive complexity, *AIP Conference Proceedings*, **1558(1)** (2013), 357–361.

[17] **Rakić, G., M. Tóth, and Z. Budimac,** Toward recursion aware complexity metrics, *Information and Software Technology*, **118:106203** (2020).

[18] **Savić, M., G. Rakić, Z. Budimac, and M. Ivanovic,** A language-independent approach to the extraction of dependencies between source code entities, *Information and Software Technology*, **56(10)** (2014), 1268-1288.

[19] **Savic, M., G. Rakić, Z. Budimac and M. Ivanovic,** Extractor of software networks from enriched concrete syntax trees, *AIP Conference Proceedings*, **1479** (2012), 486–489.

[20] **Stafford, J. A. and A. L. Wolf,** *A Formal, Language-Independent, and Compositional Approach to Interprocedural Control Dependence Analysis*, PhD thesis, USA, 2000.

[21] **Tóth, M. and I. Bozó,** Static Analysis of Complex Software Systems Implemented in Erlang, *Central European Functional Programming Summer School – Fourth Summer School, CEFP 2011, Revisited Selected Lectures*, Lecture Notes in Computer Science (LNCS), **7241** (2012), 451–514.

I. Bozó and M. Tóth
ELTE, Eötvös Loránd University
Budapest
Hungary
bozo_i@inf.elte.hu
toth_mi@inf.elte.hu

Z. Budimac, S. Knežev and G. Rakić
University of Novi Sad
Novi Sad
Serbia
zjb@dmi.uns.ac.rs
t16wzn@inf.elte.hu
goca@dmi.uns.ac.rs