View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidade do Minho: RepositoriUM

Support for Automatic Refactoring of Business Logic

Tiago B. Fernandes[1], António Nestor Ribeiro[2], David V. Nunes[3], Hugo R. Lourenço[3], and Luiz C. Santos[3]

[1] Dep. Informática Universidade do Minho
[2] Dep. Informática Universidade do Minho/Haslab - InescTec
[3] OutSystems, Inc.

Abstract. Software's structure profoundly affects its development and maintenance costs. Poor software's structure may lead to well-known design flaws, such as large modules or long methods. A possible approach to reduce a module's complexity is the Extract Method refactoring technique. This technique allows the decomposition of a large and complex method into smaller and simpler ones, while reducing the original method's size and improving its readability and comprehension.
The OutSystems platform is a low-code platform that allows the development of web and mobile applications that rely on a set of visual Domain-Specific Languages (DSLs). Even low-code languages when improperly used can lead to software that has maintenance issues like long methods.
Thus, the purpose of this paper is to present the research and development done to provide the OutSystems platform with a tool that automatically suggests Extract Method refactoring opportunities. The research combines program slicing techniques with code complexity metrics to calculate the best refactoring opportunities that preserve programs' functionality.
The proposed approach was tested on typical OutSystems apps and was shown to be able to reduce the overall applications' complexity.

Keywords: Refactoring · Program Slicing · Code Complexity Metrics · OutSystems

1 Introduction

The complexity, size and inconsistency of software demand increased maintenance costs and excessive investment of time for developing and managing it. Several studies point out causes that limit any system's functionality and development: Gill and Kemerer [1991] referred constraints caused by the high complexity degree; Banker et al. [1993] alerted the difficulties caused by large systems; Meyers and Binkley [2007] analyzed the problems caused by the lack of system's cohesion.

Software is constantly changed throughout its lifecycle. These changes are caused by bugs correction, new features implementation and adoption of new architectures, practices or development teams. Over time, all of these changes can lead to complex, graceless and incomprehensible modules.

Thus, the purpose of this paper is to provide the OutSystems platform with a tool that combines code complexity metrics and program slicing techniques to reduce modules' overall complexity.

1.1 Context and Objectives

OutSystems offers an alternative to traditional software development models. Its proprietary low-code platform allows rapid application delivery with minimal hand-coding. The platform along with its visual DSL help programmers to focus on the functional part of applications, and abstract themselves from implementation details, that result in an increased productivity and software quality. Despite the proven improvements provided by the OutSystems Platform, refactoring techniques may still help to maintain the balance and consistency of software projects.

The application of refactoring techniques is fundamental to moderate software complexity. Refactoring is the process of changing a software system to improve the design of the code without affecting its external behavior [Fowler et al., 2000]. However, manual refactoring is risky and thus it may imply programmers to be reluctant to its practice. Such behaviour may be due to various factors: lack of knowledge of refactoring techniques; the fact its benefits is only seen in the long term; being a complementary activity to the development of features and bug fixing; being a risky operation that may introduce bugs. Furthermore, refactoring is riskier if it is practiced in a disorganized way or without a methodical approach.

Projects developed in the OutSystems Platform may benefit from the application of some refactoring techniques, given the existence of large and complex modules. These large modules are often referred to as "Long Methods" in Object-Oriented Programming. In the OutSystems' context, an action has similar behavior to a method, and thus we refer to it as "Long Action". One of the most widely used techniques for reducing the complexity of modules is the Extract Method primitive [Fowler et al., 2000], which is characterized by extracting code from a routine to a subroutine so that the original routine's complexity reduces.

The OutSystems Integrated Development Environment (IDE), as well as other industry-established IDEs, has refactoring primitives such as the Extract Action. However, the identification of refactoring opportunities remains a process dependent on human intervention. Thus, automatic identification of refactoring opportunities is imperative, in a context where IDEs supervise and suggest changes in the code.

Refactoring applied externally and exclusively by the programmer must be assisted by the IDE itself, through the presentation of suggestions and respective benefits. The aid of IDEs in the automatic identification and application of refactoring techniques guarantees a methodical intervention which is not always guaranteed by an expert. Moreover, being the IDE the one responsible for changing the software makes the entire refactoring process faster and more efficient. The methodical application of refactoring techniques is essential, so that the system maintains its expected quality and integrity.

Thus, the purpose of this paper is to present the research and development done to provide the OutSystems platform with a tool that automatically suggests Extract Action refactoring opportunities. The research combines program slicing techniques with code complexity metrics to calculate the best refactoring opportunities that preserve programs' functionality.

2 Related Work

Given the wide complexity range of modules, it is imperative to analyze which modules are the most complex. This way, it is possible to select the most useful actions to refactor.

Complexity analysis can be addressed in two ways: quantitative analysis and structural analysis. Quantitative metrics consider program quantitative aspects such as its dimension. For example, the Lines of Code (LOC) metric measures the program's length, i.e., it counts the number of rows. This metric is often used in software development, given its simplicity to understand and apply.

On the other hand, structural metrics consider program structure, e.g., they study the number of paths of an action and the data flow carried. For instance, McCabe [1976] introduced the Cyclomatic Complexity ($CC = e - n + 2p$) metric that measures a program's structure based on the paths of the Control Flow Graph (CFG) [Allen, 1970]. Madi et al. [2013] present the metrics Total Cyclomatic Complexity (TCC) and Coupled Cyclomatic Complexity (CCC). While TCC aims at measuring the global Cyclomatic Complexity (CC) of a module, CCC intends at measuring the coupling level between modules' components.

Next, it became necessary to decompose modules into simpler ones to reduce their complexity. For that, there were studied multiple program slicing techniques that are able to extract behaviour from complex modules.

Program slicing is a technique that allows code extraction from a routine to a subroutine while reducing the complexity of the first. Slicing was originally designed to ease debugging processes [Weiser, 1981], but quickly became relevant in metrics calculation, code analysis and maintenance [Harman and Hierons, 2001, Tsantalis and Chatzigeorgiou, 2011].

Concerning runtime information, static slicing [Weiser, 1981] consists of all statements that can affect the value of a variable v and is directly associated with the total computation of a variable (complete computation slice). On the other hand, dynamic slicing [Korel and Laski, 1988] considers a specific value of the variables for a specific execution of a program to present better slices (regarding the input values). Conditioned slicing [Canfora et al., 1998] does not consider the variables' values but rather the conditions' values, i.e., whether or not a control predicate is true.

Regarding flow direction, backward slicing produces slices that contain all statements and control predicates that may affect a variable at a specific statement (slicing criterion). On the other hand, forward slicing [Bergeretti and Carry, 1985] produces slices that contain all statements and control predicates that may be affected by a variable at a specific statement.

Concerning syntax preservation, syntax-preserving slicing aims at extracting code from a routine to a subroutine by extracting statements and control predicates without changing them. On the other hand, amorphous slicing Harman and Danicic [1997] computes slices using syntactic transformations that may change the code, e.g., by changing control predicates.

Regarding slicing scope, intraprocedural slicing uses a single procedure as a boundary to compute slices, while interprocedural slicing [Horwitz et al., 1988] generates slices with respect to multiple functions calls. Intraprocedural slicing uses the Program Dependence Graph (PDG) [Ferrante et al., 1987] that represents control and data dependencies between nodes and interprocedural slicing uses the System Dependence Graph (SDG) [Horwitz et al., 1988] to represent interactions between multiple PDGs of a module.

Maruyama [2001] introduced the concept of block-based slicing to produce slices that do not use the whole procedure as region, but rather a smaller portion of the code. By limiting the slice expansion it is possible to extract a single variable in multiple ways, which result in more suggested slices.

3 The OutSystems Platform

The OutSystems Platform is a high-productivity platform intended for developing enterprise web and mobile applications. Its low-code IDE allows programmers to quickly develop applications with little effort and reduced costs resulting in a shorter Time to Market. It comprises a set of symbols, the most relevant of which are:

- The Start node indicates the beginning of the flow.
- The End node indicates the ending of the flow.
- The Server Action call node execute server side functions.
- The Aggregate node easily allows to retrieve data from a database.
- The If node executes a specific branch based on its condition.
- The For Each node iterates over a list of items.
- The Assign node which assigns a value to a variable. This node may have multiple Assignments, avoiding the creation of multiple Assigns.

Figure 1 represents, in the OutSystems platform, an adaptation done by Tsantalis and Chatzigeorgiou [2011] of a well-known refactoring example presented by Fowler et al. [2000]. An action in OutSystems is a flowchart with a single entry point and may have multiple exit points.

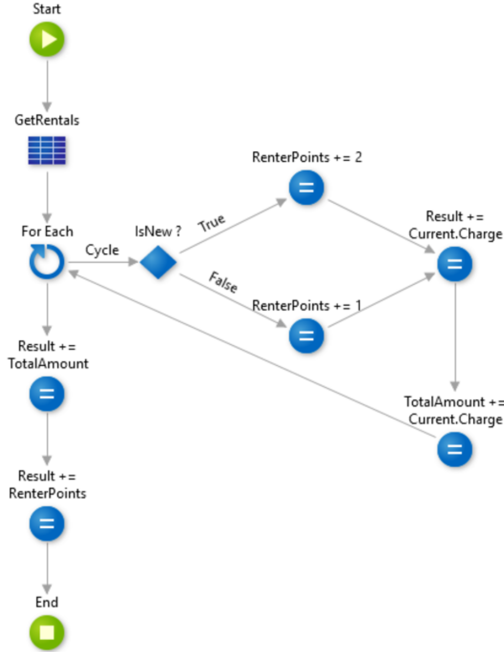


Fig. 1: Statement action in OutSystems

4 Proposed Solution

The proposed solution intends, in the first place, to evaluate the complexity of modules in order to ascertain the need of their refactoring. This step identifies the actions that are most likely to reduce their complexity after refactoring. The complexity of modules is calculated based on the number of nodes (equivalent to LOC in the OutSystems context) and the value of CC. According to Software Improvement Group (SIG), these are two relevant metrics for analyzing the quality and degree of software maintenance [Visser et al., 2016]. Both metrics are used to evaluate whether an action should be refactored by selecting the ones that would produce the best refactoring opportunities. The used threshold values were the ones defined in SIG [Visser et al., 2016]: 15 nodes for LOC and 4 units of CC.

Secondly, the proposed solution aims at reducing the complexity of modules by applying the Extract Action refactoring technique. Program slicing was the technique used to identify parts of the code that compute a common variable and then extract them to a different action.

The proposed solution is based on the Tsantalis and Chatzigeorgiou [2011] proposal and presents the following properties: (i) backward slicing, (ii) static slicing, (iii) intraprocedural slicing, (iv) syntax preserving slicing and (v) block-based slicing (see Section 2). A backward slice is computed by ignoring parts of the code that do not affect the slicing criterion, while a forward slice is computed by ignoring the statements that cannot be affected by the slicing criterion [Harman and Hierons, 2001]. Since there was no evidence of any performance increase between the two, we kept with the original definition of program slicing [Weiser, 1981] (in fact, the flow direction of the traversal should not affect slice computation). Static slicing was chosen over dynamic slicing because it is a more generic proposal as it computes all possible slices independent of user input. The use of a single PDG at a time (intraprocedural slicing) was to reduce implementation efforts. Choosing syntax preservation was an in-house decision to avoid possibly changing the original program's structure. Block-based slicing was the implemented solution as it allows a greater number of refactoring opportunities, since it promotes the extraction of variable within the multiple regions of the code. In this context, the technique is used to extract parts of the business logic from an action to another to reduce the first's complexity.

Maruyama [2001] proposed block-based slicing as a means to produce slices scoped in a block-based region. A block-based region $R(B_n)$ is a set of nodes in the PDG along with control and data dependencies between them. A block-based region starts at a specific basic block and ends at the last reachable one. A basic block B_n is a sequence of consecutive nodes in the CFG without branching and are delimited by leader nodes. A leader node is (i) the first node, (ii) a join node or (iii) a node that directly follows a branch node. The Start and the End nodes do not belong to any basic block. Figure 2 represents the basic blocks of the action presented in Figure 1.

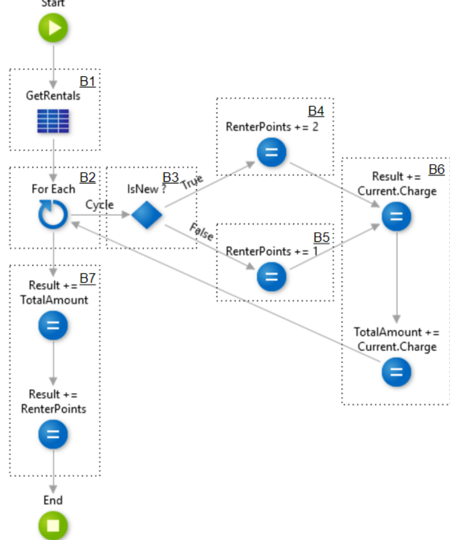


Fig. 2: Basic blocks of the CFG

Maruyama defines a block-based slice $S_B(n, v, B_n)$ as a backwards static slice that contains all statements that may affect the value of slicing criterion $C = (n, v)$ (given node n and variable v) within region $R(B_n)$ (Equation 1).

$$\begin{aligned} S_B(n, v, B_n) &= \{m \in N(G) \mid m \to^* n \in E_B(R(B_n))\}, v \in Def(n) \\ S_B(n, v, B_n) &= \bigcup_{m \in M(n,v,B_n)} S_B(m, v, B_n) \\ M(n, v, B_n) &= \{m \in N(G) \mid m \to^v n \in E_B(R(B_n))\}, v \in Use(n) \end{aligned} \tag{1}$$

Algorithm 1 represents the computation of the nodes of the block-based slice, which are nodes that may affect the value of a variable v within the region R, given the nodes and edges that belong to it.

Algorithm 1 Block-based Slice

```
1: procedure SliceNodes(node, variable, nodes, edges)
2:     result ← {}
3:     if variable is null || variable ∈ Def(node) then
4:         result ← result ∪ BackwardsTraversal(node, edges)
5:     else if variable ∈ Use(node) then
6:         result ← result ∪ BackwardsTraversal(node, edges)
7:         for each defNode ∈ IncomingDataDependencies(node) do
8:             result ← result ∪ BackwardsTraversal(defNode, edges)
       return result
```

Algorithm 2 represents the implemented backwards traversal algorithm used by Algorithm 1, given the edges that belong to block-based region R. For aesthetics

reasons, Algorithm 2 shortcuts incoming control dependencies and incoming data dependencies to "IncControlDeps" and "IncDataDeps", respectively.

Algorithm 2 Bottom-up Traversal of the PDG

```
 1: procedure BackwardsTraversal(initial, edges)
 2:     result ← {initial}
 3:     Add initial to queue
 4:     while queue is not empty do
 5:         node ← Dequeue(queue)
 6:         for each dep ∈ IncControlDeps(node) ∪ IncDataDeps(node) do
 7:             if dep ∈ edges then
 8:                 Add Source(dep) to result
 9:                 if Source(dep) is not visited then
10:                     Add Source(dep) to queue
      return result
```

As a means to extract the slice to a different action, it is needed to evaluate whether or not the nodes can be removed from the original one. For that, Maruyama defines as indispensable I_B the nodes that belong to the slice S_B but should not be removed from the action to preserve its original behaviour. Thus, indispensable nodes should exist both in the original and the created action [Tsantalis and Chatzigeorgiou, 2011]. This nodes are indispensable due to existing control or data dependencies from slice nodes S_B to non-slice nodes U_B (Equation 2).

$$\begin{aligned} &I_B(S_B, U_B, v, B) = I_{CD} \cup I_{DD}, \\ &I_{CD} = \{q \in N(m) \mid (q \in S_B(p, u, B) \vee q = p) \wedge p \in N_{CD}(S_B, U_B) \wedge u \in Use(p)\}, \\ &I_{DD} = \{q \in N(m) \mid q \in S_B(p, u, B) \wedge p \in N_{DD}(S_B, U_B, v) \wedge u \in Def(p)\} \end{aligned} \tag{2}$$

The computation of the slice nodes with at least one control dependency to non-slice nodes N_{CD} is given by Equation 3.

$$N_{CD}(S_B, U_B) = \{p \in N(m) \mid p \rightarrow_c q \in E(m) \wedge p \in S_B \wedge q \in U_B\} \tag{3}$$

The computation of the slice nodes with at least one data dependency to non-slice nodes N_{DD} is is given by Equation 4.

$$N_{DD}(S_B, U_B, v) = \{p \in N(m) \mid p \rightarrow_d^u q \in E(m) \wedge u \neq v \wedge p \in S_B \wedge q \in U_B\} \tag{4}$$

The aforementioned equations use the following definitions:

$\mathbf{N(m)}$ represents the nodes of the PDG.
$\mathbf{E(m)}$ represents the dependencies of the PDG.
$\mathbf{S_B}$ represents the slice nodes.
$\mathbf{U_B}$ represents the remaining nodes $U_B = N(m) \backslash S_B$.
$\mathbf{Use(p)}$ is the set of variables used by node p.
$\mathbf{Def(p)}$ is the set of variables defined by node p.
$\mathbf{N_{CD}}$ is the set of nodes $p \in S_B$ with a control dependency to a node $q \in U_B$.

N_{DD} is the set of nodes $p \in S_B$ with a read after write (RAW) data dependency (due to variable v) to a node $q \in U_B$.
IncomingDataDependencies(p) is the set of RAW data dependencies from node $q \in N(m)$ to node p.
IncomingControlDependencies(p) is the set of control dependencies from node $q \in N(m)$ to node p.

The action in Figure 1 was subject to the implemented algorithm, producing multiple slicing opportunities for each variable within each region. Figure 3 represents one of the slicing opportunities generated by the algorithm. It illustrates the block-based slice for variable "RenterPoints" at the block-based region $R(B_2)$ delimited by the dashed box. Circled in green are the removable nodes and in red are the indispensable nodes that will be duplicated after slicing.

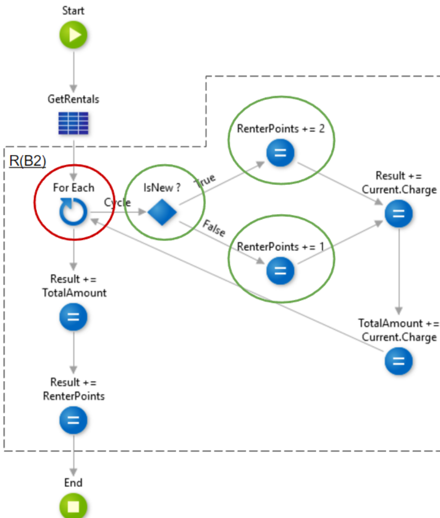


Fig. 3: Block-based slice of "RenterPoints" in $R(B_2)$

However, Maruyama's algorithm does not guarantee behaviour preservation related with the duplication of statements. Tsantalis and Chatzigeorgiou [2011] presented a set of rules regarding behaviour preservation, the most relevant of which for this paper are the preservation of existing anti-dependencies (write after read) and preservation of existing output-dependencies (write after write). These dependencies should be preserved if they start in a node that will exist in the original action after slicing and end in a node that will be removed from the original action after slicing. If not, the order or execution will be changed after extracting the slice [Tsantalis and Chatzigeorgiou, 2011], affecting the action's original behaviour.

5 Case Study

The selected case study is composed by 9 modules developed in the OutSystems platform for some company's purposes. These modules provide employees with means of regression tests analysis, support for new incomers' integration processes, employee information listing, wage premium management and holiday scheduling.

The modules were chosen for their characteristics that are adverse of software quality as they were maintained in rotating team environments. Currently, the modules consist in 1643 actions that contribute, generally, to the system's complexity. The conditions under which the modules were developed and maintained throughout their life cycles are prone to design flaws such as Long Actions.

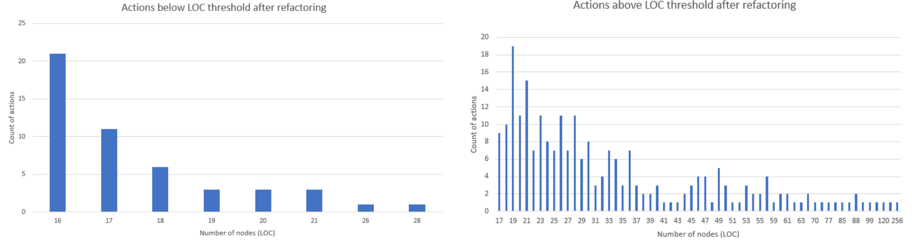
As previously mentioned in Section 4, the selected threshold values are: 15 nodes for LOC and 4 units of CC, i.e., the selected actions for refactoring are the ones whose LOC or CC is higher than the corresponding threshold value.

From the total 1643 identified actions, 331 of them (≈20%) have more than 15 nodes. Around 88% of the actions that are above the LOC threshold (290 actions out of the 331 evaluated actions) produced at least one refactoring opportunity. After refactoring, 49 actions (≈17% of the 290 actions that were above the LOC threshold) reduced their complexity to a point where refactoring is no longer needed, i.e., dropped below the LOC threshold. The remaining 241 actions (≈83% of the 290 actions that were above the LOC threshold) reduced their complexity but are still above the LOC threshold. Table 1 sums up this analysis, given the LOC threshold value at 15 nodes.

Table 1: LOC threshold analysis

	Number of actions	**Percentage (%)**
Total number of actions	659	100%
Number of actions above the LOC threshold	331	20%
Number of actions above the LOC threshold with at least one refactoring opportunity	290	88%
Number of actions below LOC threshold after refactoring	49	17%
Number of actions above LOC threshold after refactoring	241	83%

The 49 actions that once were considered complex but no longer are have a number of nodes mostly ranged between 16 and 20 nodes, as Figure 4a suggests. This may infer that the actions that dropped below the LOC threshold are relatively close to it. The remaining 241 actions that are still complex after refactoring have a number of nodes ranged between 17 and 256 nodes, as Figure 4b suggests. Thus, one can infer that 1 single application of this technique may not be enough. As a matter of fact, 234 actions (≈97% of the actions that are still complex after refactoring) produced at least one more valid slice.



(a) Nodes that dropped below LOC threshold after slicing

(b) Nodes that remained above LOC threshold after slicing

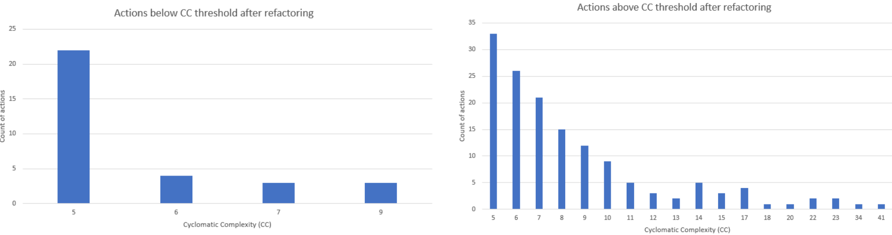
Fig. 4: LOC distribution after slicing

From the total 1643 identified actions, 199 of them (≈12%) have a cyclomatic complexity higher than 15 nodes. Around 89% of the actions that are above the CC threshold (178 actions out of the 199 evaluated actions) produced at least one refactoring opportunity. After refactoring, 32 actions (≈18% of the 178 actions that were above the CC threshold) reduced their complexity to a point where refactoring is no longer needed, i.e., dropped below the CC threshold. The remaining 146 actions (≈82% of the 178 actions that were above the CC threshold) reduced their complexity but are still above the CC threshold. Table 2 sums up this analysis, given the CC threshold value at 4 units.

Table 2: CC threshold analysis

	Number of actions	**Percentage (%)**
Total number of actions	659	100%
Number of actions above the CC threshold	199	12%
Number of actions above the CC threshold with at least one refactoring opportunity	178	89%
Number of actions below CC threshold after refactoring	32	18%
Number of actions above CC threshold after refactoring	146	82%

The 32 actions that once were considered complex but no longer are have a CC value mostly ranged between 5 and 9 units, as Figure 5a suggests. This may infer that the actions that dropped below the CC threshold after slicing are relatively close to it. The remaining 146 actions that are still complex after refactoring have a CC value ranged between 5 and 41 units, as Figure 5b suggests. Thus, one can infer that 1 single application of this technique may not be enough. As a matter of fact, 115 actions (≈79% of the actions that are still complex after refactoring) demonstrated the ability to produce at least one more valid slice.



(a) Nodes that dropped below CC threshold after slicing

(b) Nodes that remained above CC threshold after slicing

Fig. 5: CC distribution after slicing

6 Conclusions and Future Work

The implemented solution aims at identifying Extract Action refactoring opportunities that best reduce the complexity of modules. As such, the block-based slicing technique presented by Maruyama [2001] was used to extract fragments of code that compute a common variable.

In order to accelerate the study of this approach, some limitations are presented as follows: preparation actions are not refactored because some nodes may declare shared variables; Assign nodes with multiple Assignments are split up in multiple Assign nodes with one single Assignment, i.e., the algorithm can generate slices that remove nodes that were initially not visible to the programmer (although one Assign node with multiple Assignments has computational complexity similar to multiple Assign nodes with a single Assignment).

This work supports that low-code platforms still need refactoring primitives. Furthermore, Section 5 (Case Study) proves that the implemented algorithm reduces the complexity of OutSystems modules. Also, the evaluation was based on the analysis of applications whose complexity is similar to ones developed in an Information Technology department and can be considered as good case study examples. Thus, it is emphasized that this study employed on real modules, i.e., endowed with actions with a good diversity of code structure.

Section 5 (Case Study) confirms the need of a tool that automatically identifies improvements in the OutSystems code to maintain the balance and consistency of its projects, by assisting the programmer with valuable refactoring opportunities. Despite that, it is intended that the programmer can qualitatively evaluate the suggestions presented, adding a subjective factor to the analysis.

As shown in Section 5 (Case Study), a single Extract Action application had limited impact on the overall complexity of the modules, as 17% $\sim$ 18% of actions dropped below the complexity threshold. However, after the application of one Extract Action, the vast majority of the refactored actions can still be refactored, which may induce the possibility to incrementally extract different variables. Thus, it is intended that the algorithm suggests multiple and successive refactorings opportunities to take greater advantage of this technique.

The OutSystems platform allows its actions to have multiple output values for a single routine. Thus, it is intended that the algorithm supports multiple variable extraction. This is relevant in situations where there is a high intersection level in slices of different variables. That is, nodes that were once indispensable could be removed from the original action. Besides that, in some situations, it would no longer be necessary to create two subroutines to remove the computation of two variables. Furthermore, this approach leads to having more or even better refactoring opportunities available.

Bibliography

Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970. ISSN 0362-1340.

Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. Software complexity and maintenance costs. *Commun. ACM*, 36(11):81–94, November 1993. ISSN 0001-0782.

Jean-franc Bergeretti and Bernard A. Carry. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 7:37–61, 1985.

Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(1112):595 – 607, 1998. ISSN 0950-5849.

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987. ISSN 0164-0925.

Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. 2000.

Geoffrey K. Gill and Chris F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE Trans. Softw. Eng.*, 17(12):1284–1288, December 1991. ISSN 0098-5589.

Mark Harman and Sebastian Danicic. Amorphous program slicing. In *Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, WPC '97, pages 70–, Washington, DC, USA, 1997. IEEE Computer Society.

Mark Harman and Robert Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001. ISSN 1529-7950.

S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Not.*, 23(7):35–46, June 1988. ISSN 0362-1340.

Raghavan Komondoor and Susan Horwitz. Semantics-preserving procedure extraction. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 155–169, New York, NY, USA, 2000. ACM.

B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, October 1988. ISSN 0020-0190.

Ayman Madi, Oussama Kassem Zein, and Seifedine Kadry. On the improvement of cyclomatic complexity metric. *International Journal of Software Engineering and its Applications*, 7(2):67–82, 2013. ISSN 17389984.

Katsuhisa Maruyama. Automated method-extraction refactoring by using block-based slicing. *SIGSOFT Softw. Eng. Notes*, 26(3):31–40, May 2001. ISSN 0163-5948.

T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, July 1976. ISSN 0098-5589.

Timothy M. Meyers and David Binkley. An empirical study of slice-based cohesion and coupling metrics. *ACM Trans. Softw. Eng. Methodol.*, 17(1):2:1–2:27, December 2007. ISSN 1049-331X.

Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *J. Syst. Softw.*, 84(10):1757–1782, October 2011. ISSN 0164-1212.

Joost Visser, Sylvan Rigal, Rob van der Leek, Pascal van Eck, and Gijs Wijnholds. *Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code*. O'Reilly Media, Inc., 1st edition, 2016.

Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.