COSMIC Functional Size Automation of Java Web Applications Using the Spring MVC Framework

Abdelaziz SAHAB[a] and Sylvie TRUDEL[a][0000-0002-4983-1679]*

[a] Université du Québec à Montréal, Montreal, QC, Canada
sahab.aziz@gmail.com; trudel.s@uqam.ca

Abstract. Functional size measurement provides a solid basis for estimating costs and maintaining good governance during software project lifecycle. As measuring manually is more labor intensive, costly and error prone, automating the measurement process has become a priority for researchers and practitioners throughout the world. In order to measure functional size based on the COSMIC method, this work targets to automate the functional size measurement from software source code, and more particularly from Web Java applications using the Spring Web MVC framework. This paper reports on activities carried out in order to meet the following four objectives: 1) Reduce measurement effort with an accuracy of more than 90% compared to experienced practitioner's measurement; 2) Obtain functional size any time during the software lifecycle; 3) Offer a reusable and modular solution; and 4) Publish the solution as open source software. To do this, we followed a five-phases methodology: 1-Start-up; 2-Prototyping; 3-Realization; 4-Evaluation; and 5-Publication. This methodology made it possible to create the CFP4J Library and publish it as open source software on the GitLab source code repository. In the current state of the CFP4J Library, the four objectives have been achieved. The contribution of this paper is in the definition of mapping rules from code and a publicly available software library to automate COSMIC functional size of Web Java applications that use the Spring MVC framework. These rules and this library can be expanded to other technologies.

Keywords: COSMIC, Automation, Spring MVC, Functional Size.

1 Automating Functional Size Measurement

The rivalry and competition between organizations in the software development field continues to increase. And to maintain a place in the market, each organization relies on its internal information when making a decision. Functional size measurement has become a useful tool for any organization that wants to be among the leaders in its business domain. Measuring functional size makes it easier to estimate costs, improve productivity (effort/size relationship), benchmark with other organizations and keep good governance on the project [1].

1.1 Standardized Functional Size Methods

At the time of writing this paper, there was five ISO standards conforming to ISO/IEC 14143 [2] for the measurement of software functional size:

- International Function Point Users Group (IFPUG) [3];
- Mark II (MkII) [4];
- Netherlands Software Measurement Association (NESMA) [5];
- Finland Software Measurement Association (FiSMA) [6];
- Common Software Measurement International Consortium (COSMIC) [7];

While the first four methods are known to be "first generation" functional size methods (FSM), the COSMIC method [8] is the only one known as a "second generation" type of method (i.e. principle-based, domain independent, open standard, etc. [9]), able to measure various kinds of software (information systems, Web apps, Mobile apps, embedded real-time, SOA components, etc.) For this reason, COSMIC was chosen as a basis to develop a library that automates functional size measurement from Java source code, more specifically the Spring Web MVC framework.

The COSMIC method is based on data movements within a functional process. A functional process is an elementary part of Functional User Requirements (FUR: a subset of user needs; requirements that describe what the software should do, in terms of tasks and services) [2][8] of the software to be measured. There are four types of data movements: Entry, eXit, Read, and Write (EXRW). Data movements are related to a data group being used or produced by a functional process.

FUR can be derived from software engineering artifacts from analysis and design phases (before the software existed) or can be derived from the software artifacts once developed, such as the software code, data dictionary, or input-output elements (interface definitions, screens and reports).

1.2 Problems with manual measuring

Manual FSM encounters two main problems. First, the measurement effort can be significant, especially for less experienced measurers [10]. Although the measurement effort decreases a lot with experience, this effort can be considered as a hurdle when the requirements prove to be ambiguous or incomplete because it forces the measurers to go and ask questions to the project members, if they are still available. Second, inexperienced measurers tend to make measurement errors, mainly due to the ambiguities of the requirements [10].

These two problems have a major impact on the indicators of a software project. In this context, the automation of the measurement of functional size has become a research target to overcome these problems. But measuring from written requirements provides the theorical size of a software and the actual size of the developed software should be measured or confirmed once that software is delivered. This can be done from confirmed delivered requirements, or from the actual code.

The main reason why an organization would be interested to measure from the software code is to first develop a baseline of the relationship between size and effort in the form of an estimation model based on functional size. Also, size measured from code

reflects what has been delivered with more accuracy than from incomplete, ambiguous or inadequate requirements, allowing some form of requirements-based sizing assessment.

2 Related work: known automation alternatives to manual measuring

Several tools exist that automate FSM from different inputs or using various techniques. From textual requirements: Based on Natural Language Processing (NLP) or Artificial Intelligence (AI), tools such as ScopeMaster detects data movements from FUR textual descriptions in English [11]; Supervised text mining allows structuring information contained in a document to extract ambiguities and FUR. Then, unsupervised text clustering allows for identification of functional processes and data groups [12].

These techniques or tools only work with English texts that must be clear, which may not be that common in the industry.

From Unified Modelling Language (UML) diagrams: Several studies used sequence diagrams can be automatically interpreted to provide FSM, where messages between objects or classes represent data movements [13] [14] [15]. Many UML editors are not free. Quite often, software teams will draw their diagrams on a white board and take a picture with a mobile phone, which image is stored in the product repository, not using any UML editor [16]. Even when UML were defined, they are not always up-to-date, or they cover only a small part of the software.

From reverse engineering of the source code, many FSM automation tools have been published. Akca and Tarhan [17] have published the "measurement library" from a 3-layers Java enterprise application, which was semi-automated as some manual transformations were required on the source code to call upon the library methods. They obtained a precision of 92% compared to manually done FSM, along with FSM cost reduction of 280%. Later, the "measurement library" was updated with a static code installer resulting in a 94% precision with a reported effort reduction of 97% [18]. Sag and Tarhan [19] proposed the "COSMIC Solver" to measure Java business applications while being executed. COSMIC Solver required that manual transformations be made to source code or binary code using aspect-oriented programming. During execution, the tool would generate UML sequence diagrams from which the FSM is done. A more stable version of the "COSMIC Solver" was later published where an open-source Web application using Java Swing was measured with a precision of 77% [19]. The JavaCFP was published as a NetBeans plugin which analyzes the Java source code and generates the COSMIC FSM results with a precision of 100% [20]. Unfortunately, the JavaCFP is not listed as an official NetBeans plugin* and is tightly coupled with NetBeans.

We were not able to find trace of these tools while searching on popular open-source repositories (i.e. GitHub, Bitbucket, Sourceforge, and GitLab). The next step would be to try contact authors of these tools and ask them to provide a link to their tool, if publicly available.

* http ://plugins.netbeans.org/PluginPortal/

Based on these sizing automation publications, we wanted to contribute to COSMIC FSM automation by targeting a technology in which sizing was not yet automated, and the Spring MVC framework became our first choice, mainly because the main author had experience developing software with it.

3 Methodology to develop an automated measurement library

3.1 Definition of our project

We wanted to develop a tool to automate COSMIC FSM for Java applications using the Spring MVC framework, where measurement would be done from the source code. Knowing the size of several released applications and comparing this size to the effort that was required to develop them is a good way to feed an organization's productivity baseline with very little effort compared to manual measurement. Our objectives were:

1. Reduce the measurement effort with an accuracy of more than 90% compared to the measurement of an experienced practitioner;
2. Obtain the functional size measurement any time during the software lifecycle;
3. Offer a reusable and modular solution (a library) that could be integrated with almost any source code analysis tool; and
4. Publish the solution as an open source software.

Since this project was developed as a master's degree capstone project, it was important to limit its scope. Therefore, it was decided to measure applications having the following characteristics: Java version 5 or higher; Web applications using Spring Web MVC framework, version 5.x.x or above, since this framework is the most popular for Java Web applications[*]; for data access, the library Spring Data is used.

3.2 A five-phases approach

Our software library was developed applying a five-phases approach as follows:

Start-up: getting prepared to develop the solution, including justifying technology choices (Java, Spring Web MVC, and GitLab as the open-source code repository), installing the development environment (Eclipse IDE, Maven, and Git) and selecting an open-source sample Java application ("Spring PetClininc Sample[†]") to test our initial solution, which would also be manually measured with COSMIC.

Prototyping our evolutive solution: the aim was to experiment our early solution with the selected open-source sample Java application, which required to identify and implement mapping rules and procedures from Spring MVC source code to COSMIC components (see section 4.1).

Implementation: formalizing the design of our CFP4J library, refining it along with mapping rules, improving our code quality, and perform testing with other identified open-source Java Spring Web MVC applications.

---

[*] According to jetbrains.com, accessed in April 2019.

[†] https://github.com/spring-projects/spring-petclinic

Evaluation: validating our solution with two other Spring Web MVC application with the aim of refining the mapping rules to increase accuracy of results.

Publication: publishing our library as an open-source solution on GitLab, including a developer's guide in GitLab's wiki pages to ensure any developer who is new to the project would be able to rapidly contribute to evolve the CFP4J library*. This paper also serves as a means to communicate its existence to the community.

4 The resulting automated measurement library

4.1 FSM Rules and Procedures

As written on the Java T Point website, "A Spring MVC is a Java framework which is used to build web applications. It follows the Model-View-Controller design pattern." Specific Spring MVC methods related to the model, the view or the controller should then be linked with specific COSMIC data movements.

In order to apply the COSMIC method adequately, it was important to ensure all phases of the COSMIC method were applied:

1. Measurement strategy phase: the purpose is to measure automatically the functional size of a Spring Web MVC application which scope include all .java files. Functional users other than human must be identified while measuring. The level of granularity relates to all methods in Java classes being noted @Controller as they contain the triggering Entries of functional processes, at least those triggered by a human functional user.
2. Mapping phase: the functional processes correspond to any method being noted as @PostMapping, @GetMapping or @RequestMapping within an @Controller class. Objects of interest (OoI) correspond to a Java class representing a business domain entity within the software. Data groups correspond to data within the view (e.g. graphical interface). As it is a Web application, the view is coded in HTML language, which does not fall within the scope of our library analyzing Java classes, but the Spring MVC framework is responsible for mapping the view's attributes into a data group corresponding to a Java class presented as an input parameter to a method of a controller type Java class, which allows identification of data groups and data movements. Data attributes are any attribute of an entity. Data movements identification and mapping was defined thoroughly in Table 1 as a set of twelve mapping rules (MR).
3. Measurement phase: Counting one COSMIC Function Point (CFP) per data movement, avoiding counting multiple occurrences of the same data movement on the same data group within any single functional process.

Table 1 provides the list of mapping rules that were implemented for the Spring MVC framework. These rules can be expanded to other technologies. It was necessary to layout an inventory of Spring MVC methods to analyze which movement type each method represents in its context.

* To access the CFP4J library refer to https://gitlab.com/asahab/cfp4j

Table 1. Data movement mapping rules.

#	Mapping rule definition
MR01	Any OoI found among the input parameters of the method with one of the annotations @PostMapping, @GetMapping, @RequestMapping or @ModelAttribute is considered an **Entry** data movement.
MR02	All input parameters having the annotation @RequestParam of a method having one of the annotations @ModelAttribute, @PostMapping, @GetMapping or @RequestMapping are considered as a single **Entry** data movement, which represents criteria of a search filter. A search criterion is excluded from the filter if it represents an attribute of an OoI found among the input parameters of the method.
MR03	Input parameters with the annotation @PathVariable of a method having one of the annotations @ModelAttribute, @PostMapping, @GetMapping or @RequestMapping are considered as an **Entry** data movement representing an attribute of an OoI which is not among the input parameters of the method.
MR04	Input parameters having the type *MultipartFile* of a method having one of the annotations @ModelAttribute, @PostMapping, @GetMapping or @RequestMapping are considered as an **Entry** data movement representing an attribute of an OoI.
MR05	Each **Entry** data movement of a method having one of the annotations @PostMapping, @GetMapping or @RequestMapping must not be an **eXit** data movement of a method having @ModelAttribute annotation.
MR06	If at least one **Read** data movement or a **Write** data movement has been recorded, and no **Entry** data movement has been associated with an input parameter of a method having one of the annotations @ModelAttribute, @PostMapping, @GetMapping or @RequestMapping, then an **Entry** data movement representing a trigger for this functional process must be added.
MR07	If a method with one of the annotations @PostMapping, @GetMapping or @RequestMapping has at least one input parameter annotated with @Valid or of type BindingResult or a method Delete was called during the execution of this method, then an **eXit** data movement is associated with the output message corresponding to the validation of the input parameter or to the confirmation of the deletion.
MR08	Each added OoI as a new element to one of the instances *Map*, *Model*, *ModelMap* or *ModelAndView* in the body of a method having one of the annotations @ModelAttribute, @PostMapping, @GetMapping or @RequestMapping, is considered as an **eXit** data movement. The OoI occurrence must not be a local occurrence (e.g. new ObjectInterest ()).
MR09	Each method with the annotation @ResponseBody and one of @PostMapping, @GetMapping or @RequestMapping annotations is a method returning an OoI which is considered as an **eXit** data movement.
MR10	Each call to one of the *findById*, *existsById*, *findAll*, *getOne*, *count* or *findAllById* methods of a Spring Data Repository class, recursively from a method having one of the annotations @ModelAttribute, @PostMapping, @GetMapping or @RequestMapping, is considered as a **Read** data movement.
MR11	Each call to one of the methods *save*, *saveAll*, *saveAndFlush*, *deleteInBatch*, *deleteAllInBatch*, *deleteById*, *delete* or *deleteAll* of a Spring Data Repository class, recursively from a method having one of the annotations @ModelAttribute, @PostMapping, @GetMapping or @RequestMapping, is considered as a **Write** data movement.
MR12	Each call to a custom method of a class implementing the Spring Data Repository class, recursively from a method having one of the annotations @ModelAttribute, @PostMapping, @GetMapping or @RequestMapping, is considered a data movement. If this method has a return type, then the data movement is **Read**. On the contrary (void), the data movement is **Write**.

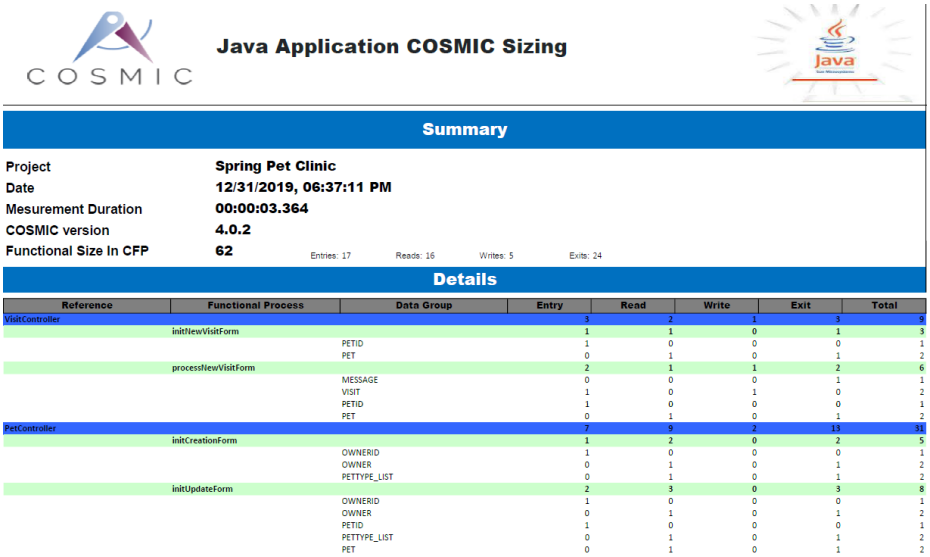
In order to avoid counting multiple occurrences of the same movement on a single data group within a functional process, some processing was needed. After all mapping rules are applied, these duplicated data movements are discarded.

To test the CFP4J library integration, a sample module called "CFP4J-Application" was implemented and published on the GitLab repository (for more details see the developer's guide).

Once the test processing is complete, the user obtains a JSON file with measurement strategy and measurement details,

as well as an Excel spreadsheet which is displayed on screen (see Fig.1).

No post-processing is necessary once the CFP4J has completed its execution.



COSMIC

Java Application COSMIC Sizing

Summary

Project	**Spring Pet Clinic**
Date	**12/31/2019, 06:37:11 PM**
Mesurement Duration	**00:00:03.364**
COSMIC version	**4.0.2**
Functional Size In CFP	**62** Entries: 17 Reads: 16 Writes: 5 Exits: 24

Details

Reference	Functional Process	Data Group	Entry	Read	Write	Exit	Total
VisitController			3	2	1	3	9
	initNewVisitForm		1	1	0	1	3
		PETID	1	0	0	0	1
		PET	0	1	0	1	2
	processNewVisitForm		2	1	1	2	6
		MESSAGE	0	0	0	1	1
		VISIT	1	0	1	0	2
		PETID	1	0	0	0	1
		PET	0	1	0	1	2
PetController			7	9	2	13	31
	initCreationForm		1	2	0	2	5
		OWNERID	1	0	0	0	1
		OWNER	0	1	0	1	2
		PETTYPE_LIST	0	1	0	1	2
	initUpdateForm		2	3	0	3	8
		OWNERID	1	0	0	0	1
		OWNER	0	1	0	1	2
		PETID	1	0	0	0	1
		PETTYPE_LIST	0	1	0	1	2
		PET	0	1	0	1	2

Fig. 1. Example of Excel spreadsheet output on screen a CFP4J result.

4.2 Solution internal quality

As an open-source project, we considered important to ensure maintainability and usability of the CFP4J library. The CFP4J library consists of 1,343 source lines of code, 18 classes, 65 methods, and 61 automated unit tests with a coverage of 79.5%. To verify and validate these quality characteristics, SonarQube was used to analyze the CFP4J source code to detect any code smell or flaw. None were found.

5 Evaluation of the CFP4J Library with three applications

Our approach consists of comparing the manually counted functional size of three software applications with the automated FSM using the CFP4J library. The aim was to

quantify the accuracy in terms of percentage of automatically counted data movements over manually counted data movements. Also, the measurement effort was quantified for manual measurement and for the required setup and execution of the CFP4J library.

Every difference between the manual count and the automatic count was analyzed by comparing the application source code, the detailed manual count and the detailed automatic count in order to understand the reason behind that difference. In most case where the manual count was smaller, it was because there were some data movements that could not have been seen from executing these applications, in which case the automatic count was more accurate. In few cases where the manual count was larger than the automatic count, the analysis revealed that two different functional processes were actually implemented using the same code, which in turn revealed a design flaw within the code as these two functional processes were likely to evolve differently.

Table 2. Physical and functional sizes of the selected Java Spring Web MVC applications.

Application	#SLOC	# Java classes	# CFP Manual	# CFP Automatic	Accuracy
Spring PetClinic	960	24	56	62	90,3%
Mini ToDo Management	374	10	17	16	93,7%
Java Blog Aggregator	4,140	69	172	173	99,4%
Total:	5,474	103	245	251	97,6%

Note 1: Accuracy is calculated as the absolute difference between manual and automatic counts, divided by the automatic count then subtracted from 1 to obtain the percentage.
Note 2: The manual count was performed by one of the authors and verified by the other author. Both authors are certified COSMIC measurers.

Manually measuring these three applications took between 1 and 6 hours, with an average varying from 1.94 to 3.75 minutes per CFP. Manual measurement effort of these applications includes installing them on the measurer computer and executing them in order to go through every functionality, while trying our best not to forget any. Preparing the environment to include the CFP4J library only take 10 minutes to do. Once integrated, using the library took between 3 to 6 seconds to execute. The FSM automation using CFP4J reduced the measurement effort by 98% on average.

6 Limitations and future work

Technological scope has been voluntarily limited. The CFP4J library needs to support other libraries to communicate with a database and with other third-party services applying the HTTP protocol. The effort required to expand the CFP4J library was way beyond the effort requirement of a 6-credits capstone project. Other projects in a near future could expand the CFP4J library to include communication with other protocols and services.

Another limitation is the fact that not all software code is compliant with the recommended Java coding convention and expected MVC architecture. For that reason, there is a possibility that a developer has been developing in such a way that the defined mapping rules may be insufficient.

In order to improve CFP4J, testing it with more applications is required, not only to test different code structures but also to test sizing larger software applications. More testing may result in added or modified mapping rules.

CFP4J does not provide a manual calibration function. Manual calibration could be useful in the future to handle different code structures.

Also, future work should consider integrating with libraries communicating with components and databases such as JPA, Hibernate, and Spring JDBC Template, and with third-party services using HTTP protocol such as Apache HTTP-Client.

An expanded version of CFP4J could include publishing it on a Maven repository to facilitate its integration for any project wishing to use CFP4J. The community should consider expanding CFP4J to other Java application types such as JSF and Java Swing. CFP4J could also be implemented as a SonarQube plugin, providing the functional size as the entire code is being analyzed on a daily basis, which would provide the advantage of monitoring functional size growth over time.

7 Conclusion

With the development of the CFP4J library, all of our four objectives have been achieved:

- Reduce the measurement effort with an accuracy of more than 90% compared to the measurement of an experienced practitioner → The average accuracy was of 97.6% with an average reduction of effort of 98%.
- Obtain the functional size measurement any time during the software lifecycle → CFP4J can be used anytime during the development or after the release of an application.
- Offer a reusable and modular solution (a library) that could be integrated with almost any source code analysis tool → A JAR file is available and can be integrated as a dependence of an application source code.
- Publish the solution as an open source software → This objective achieved through https://gitlab.com/asahab/cfp4j.

The contribution of this paper is in the definition of mapping rules from code and a publicly available software library to automate COSMIC functional size of Web Java applications that use the Spring MVC framework. These rules and this library can be expanded to other technologies.

Acknowledgements

While validating the CFP4J library, few points needed to be clarified and questions were asked to COSMIC Measurement Practice Committee members. The authors are grateful to those who have promptly answer, namely Arlan Lesterhuis (Netherlands) and Jean-Marc Desharnais (Canada).

References

1. Trudel, S.: Course notes: Measures and software development, summer 2018, UQAM, Montreal, Canada (2018).
2. ISO/IEC 14143-1: Information technology — Software measurement — Functional size measurement — Part 1: Definition of concepts, 2nd edn, International Organization for Standardization (ISO), Geneva, Switzerland (2007).
3. ISO/IEC 20926: Software and systems engineering - Software measurement - IFPUG functional size measurement method 2009, 2nd edn, ISO, Geneva, Switzerland (2009).
4. ISO/IEC 20968: Software engineering - Mk II Function Point Analysis - Counting Practices Manual, 1st ed., Dec-2002, ISO, Geneva, Switzerland (2002).
5. ISO/IEC 24570: Software engineering - NESMA functional size measurement method - Definitions and counting guidelines for the application of function point analysis, 2nd ed., ISO, Geneva, Switzerland (2018).
6. ISO/IEC 29881: Information technology - Systems and software engineering - FiSMA 1.1 functional size measurement method, 1st ed., ISO, Geneva, Switzerland (2010).
7. ISO/IEC 19761: Software engineering - COSMIC: a functional size measurement method, 2nd ed., ISO, Geneva, Switzerland (2011, reviewed and confirmed in 2019).
8. Abran, A. et al. (2020). COSMIC Measurement Manual for ISO 19761, Part 1: Principles, Definitions & Rules, version 5.0, March 31st, 2020. Available from https://www.cosmic-sizing.org, accessed 2020/06/10.
9. COSMIC (2020). Second Generation Functional Size Measurement by Design, Available from https://cosmic-sizing.org/cosmic-sizing/functional-size-measurement/second-generation/, last accessed 2020/08/30.
10. Trudel, S. and Abran, A.: Functional Size Measurement Quality Challenges for Inexperienced Measurers. In: International Workshop on Software Measurement (IWSM-Mensura) 2009. pp. 157–169, Springer, Heidelberg (2009).
11. Ungan, E, Hammond, C. Abran, A.: Automated COSMIC Measurement and Requirement Quality Improvement Through ScopeMaster® Tool. In: IWSM-Mensura 2018. Beijing, China (2018).
12. Hussain, I., Ormandjieva, O., Kosseim. L.: Mining and Clustering Textual Requirements to Measure Functional Size of Software with COSMIC. Software Engineering Research and Practice, pp. 599–605, (2009).
13. Bévo, V., Lévesque, G., Abran, A.: UML notation for functional size measurement method, In: IWSM 1999, Lac-Supérieur, Canada (1999).
14. Fehlmann, T.M., Kranich, E.: COSMIC functional sizing based on UML sequence diagrams. MetriKon, Kaiserslautern (2011).
15. Karim, S., Liawatimena, S., Trisetyarso, A. et al.: Automating functional and structural software size measurement based on XML structure of UML sequence diagram. In: IEEE International Conference on Cybernetics and Computational Intelligence 2017 (CyberneticsCom). IEEE, pp. 24–28. (2017).
16. Christopher, F.: Course notes on Advanced software design, winter 2017. Montreal, Canada (2017).
17. Akca, A.A., Tarhan, A.: Run-time measurement of COSMIC functional size for java business applications: Initial results. In: IWSM-Mensura 2012. IEEE, pp.226–231. (2012).
18. Gonultas, R., Tarhan, A.: Run-time calculation of COSMIC functional size via automatic installment of measurement code into Java business applications. 41st Euromicro Conference on Software Engineering and Advanced Applications. IEEE, pp.112–118. (2015).

19. Sag, M.A., Tarhan, A.: Measuring COSMIC software size from functional execution traces of Java business applications. IWSM-Mensura 2014, IEEE, pp.272–281. (2014).
20. Chamkha, N., Sellami, A., & Abran, A.: Automated COSMIC Measurement of Java Swing Applications throughout their Development Life Cycle. In: WSM-Mensura, pp. 20-33. (2018).