Towards Universal COSMIC Size Measurement Automation *

Hassan Soubra[1], Yomna Abufrikha[1], and Alain Abran[2]

[1] German University in Cairo, New Cairo, Egypt
hassan.soubra@guc.edu.eg
yomna.abufrihka@student.guc.edu.eg
[2] École de technologie supérieure – ETS, Université du Québec, Montréal, Canada
alain.abran@etsmtl.ca

Abstract. Today there are a large number of computer programming languages, e.g., Java, C, C++, Python, to name a few. The COSMIC functional size measurement method can capture the functionality of software written in any language. Automating functional size measurement (FSM) from code allows a large number of projects to be measured in a short time. However, because of the diversity of programming languages, a specific automation tool is currently needed for each one. To address this issue, we exploit the property that once a program is translated into machine code, it becomes independent of the original language it was written in, which is a basis for designing a 'universal' automation tool. This paper proposes an approach for a 'universal' tool based on COSMIC ISO 19761 for automated measurement of software written in different programming languages. As a proof of concept, this paper presents a prototype tool based on COSMIC and MIPS, with a small-scale validation.

Keywords: COSMIC · MIPS ISA · Automation Tool · ISO 19761 · Measurement Automation.

1 Introduction

The COSMIC functional size measurement (FSM) method [1] is used to measure functional user requirements (FUR) throughout the software development process, from the requirements specification phase for estimation purposes to post-implementation analysis for productivity and bench-marking studies.

However, applying FSM procedures manually is tedious and time-consuming, which is problematic for organizations with a large number of projects to measure in a very short time, either for project estimation purposes or for productivity studies. In addition, the manual application of FSM to a very large set of source code inputs requires specialized expertise when there is a variety of languages in which the source code has been implemented.

According to the Online Historical Encyclopedia of Programming Languages [2], there are 8,945 programming languages, which fall into two categories: interpreted or scripted, such as Perl and Python, and compiled, such as C, C++, and Java. Furthermore, different programming languages produce different results when implementing the same set of requirements [3].

Automating COSMIC-based FSM requires a complete mapping between the principles of the COSMIC method and the input notation that describes the functional user requirements. Each programming language has its own notation. Hence, a specific mapping for each programming language is required for automation to become feasible. A number of COSMIC-based automation tools exist in the literature, e.g. a tool that uses the Simulink model [4] for automation from requirements models and ScopeMaster for automation from textual requirements [5, 6].

A universal tool applicable to all types of input languages would be ideal, but also challenging when measurement inputs are expressed in different programming languages. To address this issue, we exploit the property that once a program is translated into executable binary code, it becomes independent of the programming language it was written in. This property could be helpful for designing a 'universal' automation tool.

This paper presents a feasibility study of an approach to a 'universal' tool based on COSMIC ISO 19761 and MIPS to automate the measurement of software written in different programming languages. The paper is organized as follows. Section 2 presents an overview of the COSMIC – ISO 19761 measurement method. Section 3 reviews the literature on existing COSMIC based measurement automation tools. Section 4 discusses the proposed approach using MIPS for automating the COSMIC measurement method. Section 5 details the proposed automation prototype tool, including a measurement example. Finally, section 6 presents our conclusions and a discussion of future work.

2 COSMIC Overview

ISO 14143-1 specifies that an FSM method must measure software FUR. In addition, the COSMIC–ISO 19761 [7] standard proposes a generic model for software FUR that can capture the functionality of any type of software in a measurable way. From this generic model of software FUR the following observations can be made:

- Software is bounded by hardware. In the so-called "front end", software used by a human is bounded by I/O hardware, such as a mouse, keyboard, printer, and display, or by engineered devices, such as sensors or relays. In the "back end", the software is bounded by persistent storage hardware, such as a hard disk, or RAM or ROM memory.
- Software functionality is embedded within the functional flows of data groups. These data flows can be characterized by four distinct types of data movements. Two types of movement–Entries (E) and Exits (X)–allow the exchange of data with users across a boundary. Two other types of movement–

Reads (R) and Writes (W)–allow the exchange of data with the persistent storage hardware.
– Different abstractions are typically used for different measurement purposes. In real-time software, the users are typically the engineered devices that interact directly with the software, that is, the users are considered the I/O hardware. For business application software, the abstraction commonly assumes that the user is one or more humans who interact directly with the business application software across the boundary, and the I/O hardware is ignored.

As an FSM method, COSMIC is aimed at measuring the size of software based on identifiable software FUR.

3 Related Work

A number of COSMIC-based automation tools and prototypes exist in the literature. Google Scholar https://scholar.google.com/ was mainly used for the literature review. The keywords used were: "COSMIC automation tool", "COSMIC automated measurement; COSMIC functional size automation".
Table 1 presents a summary of the various tools identified in the literature.

Table 1. Tools Identified in the Literature Review

Tool	**Year**	**Author**	**Domain**	**Type**
Tool using Simulink model	2011	Soubra et al. [4]	Real-time embedded systems	Industrial tool
ScopeMaster	2018	Hammond et al. [5]	Real-time	Industrial tool
A tool for the automation of functional size measurement with RUP	2004	Azzouz and Abran [9]	Real-time	Academic tool
μcROSE	2005	Diab et al. [10]	Real-time	Industrial tool
A procedure that measures the functional size	2015	Gonultas and Tarhan [11]	Java business application	Industrial tool
A tool using SCADE model	2015	Soubra et al. [12]	Real-time embedded system	Academic tool
UML profile tool	2011	Lind and Heldal [13]	Embedded systems Component	Academic tool

Soubra et al. [4] proposed an automation tool based on the mapped rules between Simulink, a graphical programming environment for modeling, simulating

and analyzing multidomain dynamical systems, and COSMIC. They also detailed the algorithm behind the proposed tool and verified it using a three-state protocol.

Hammond et al. [5] presented ScopeMaster, the first commercial tool, which performs a COSMIC measurement on a set of free-form textual requirements in English. ScopeMaster performs several successive steps of analysis, individually and collectively, on the textual requirements in order to detect possible Objects of Interest, potential users, potential data movements and potential defects. The details of how ScopeMaster performs these techniques are proprietary and protected by a pending patent application; however, the results are fully transparent. The underlying steps include natural language processing and several modules of pattern matching.

Lind and Heldal [8] presented a tool based on COSMIC for measuring the functional size of embedded automotive software early in the development cycle using a UML profile that captures all the information needed for functional size measurement and estimating code size.

Azzouz and Abran [9] presented an exploratory approach for the automation of functional size measurement with RUP based on direct mapping between COSMIC-FFP and UML concepts and notation, where the inputs are the Rational Rose artifacts of the project to be measured. The tool provides a software size determination at three stages of the development cycle (the use case level, scenario level, and the COSMIC level). The first two levels provide early indicators of the functional size, which is then more precisely measured in the third level. This tool had a number of limitations such as the need to manually identify and add the project scope and a tag for a triggering element; furthermore, no large-scale testing had been carried out.

Another exploratory tool, called μcROSE, was proposed by Diab et al. [10]. The tool automatically measures COSMIC for Rational Rose RealTime (RRRT) models. The paper stated that μcROSE improves COSMIC measurements in two ways. First, it removes measurement variance and ensures perfect repeatability, under the assumption that capsules selected belong to the same layer and, second, because the measurement is automated it nearly eliminates measurement cost. The seven supported functionalities of μcROSE are: i) visual support of the COSMIC measurement process; ii) generation of an RRRT model in XML format; iii) extraction of RRRT entities required in the measurement process; iv) analysis of C++ code included in the RRRT model; v) identification of functional processes, data groups, and data movements; vi) calculation of COSMIC representing the functional size; and vii) aggregation and reporting of measurement results.

A procedure that automatically measures the functional size was proposed by Gonultas and Tarhan [11] for Java business applications with a user interface and three-tier architecture. The procedure was developed within a software package by 80 developers. According to the paper in order to use the package as an automation tool for a Java application, it needs to be deployed first then imported to the application.

Soubra et al. [12] developed a COSMIC functional size measurement procedure for real-time embedded systems in the aerospace domain. Their study presented an application of the proposed COSMIC FSM procedure to an aerospace system example designed in SCADE. It also included a comparison between the measurement results obtained by a manual procedure and the automated one obtained by a prototype tool developed at ESTACA. This procedure measures both management information and real-time system information, unlike traditional FSM methods. This paper illustrates the mapping between COSMIC and the SCADE model with the Roll Control a real-time embedded system as an example.

In another paper Lind and Heldal [13] proposed a UML profile tool to automate the estimation of the size of code based on COSMIC function points (CFP). They investigated the manual effort involved in estimating code size (e.g. it would require up to 2.5 person-years of effort to manually obtain the value of CFP for a Saab car). This paper provided a case study for mapping between COSMIC and a UML profile.
In conclusion, none of the proposed tools can be considered universal since they require specific types of input languages/models. The concept of a universal tool is a tool that is applicable to all types of input languages/models.

4 Proposed Approach for a Universal FSM Automation tool

FSM automation tools are all based on the input artifacts. Therefore, a tool designed for inputs implemented in C will not work for example for inputs in Java. However, portable languages such as C and Java that have different notations (syntax) are translated by a compiler into assembly language and then into binary machine instructions executable by the hardware. This section first presents an overview of MIPS followed by our proposed approach to map COSMIC to MIPS machine instructions.

4.1 MIPS Overview

MIPS [14] is a reduced instruction set computer (RISC) architecture evolved and developed by MIPS Technologies. MIPS architecture is a high performance, industry-standard architecture that provides a 32-bit (MIPS32) to 64-bit (MIPS64) range instruction set. The MIPS64 architecture is backward compatible with the MIPS32 architecture. Both the MIPS32 and MIPS64 architectures provide a privileged environment to address the needs of operating systems. Both also include provisions for adding optional components—modules of the base architecture, MIPS application-specific extensions (ASEs), user-defined instructions (UDIs), and custom coprocessors. The key concepts of the MIPS architecture are:

- Five-stage execution pipeline: fetch, decode, execute, memory-access, write-result.

- Regular instruction set, all instructions are 32-bit.
- Three-operand arithmetical and logical instructions.
- 32 general-purpose registers of 32-bits each.
- No status register or instruction side-effects.
- No complex instructions (e.g. stack management, string operations, etc.).
- Optional co-processors for system management and floating-point.
- Only the load and store instruction access memory.
- Flat address space of four gigabytes of main memory (2^{32}).
- Memory-management unit (MMU) maps virtual to actual physical addresses.

The components of MIPS architecture are:

- MIPS instruction set architecture (ISA)
- MIPS privileged resource architecture (PRA)
- MIPS modules and application-specific extensions (ASEs)
- MIPS user defined instructions (UDIs)

The most important component of the MIPS architecture within the scope of this paper is the MIPS ISA. MIPS is a RISC processor, so every instruction has the same length — 32 bits (4 bytes). These bits have different meanings according to their displacement. Table 2 shows the names of the different fields in a MIPS instruction, along with their size and their use.

Table 2. MIPS instruction fields

Name	**Size in bits**	**Used for**
Opcode	6	Specification of instruction
Register specifications	5	Addresses of registers
Register-immediate	5	Second part of opcode for RI and CP instructions
Shamt	5	Constant value for shifts
Immediate constant value	16	Immediate value for arithmetic and logical (AL) operations
Address	26	Address for jumps and procedure calls
Funct	6	The second part of an opcode for instructions

MIPS architecture supports instructions with up to three registers: s-register, t-register, and d-register; and seven types of instruction formats, as shown in Table 3.

Table 3. MIPS instruction components

Type	Reg #	Immediate	Used for
R	3	5 bits	AL and shift operations on registers
RI	1	16 bits	Branches
I	2	16 bits	AL operations with immediate values, load/stores, branches
J	0	26 bits	Unconditional branches, procedure calls
COP0	2	5 bits	Interaction with co-processor 0
Special2	3	5 bits	MIPS32 extensions
Special3	3	5 bits	MIPS32 secret instructions

4.2 Our Approach: Mapping COSMIC to MIPS

Our approach consists of two steps: Mapping COSMIC's principles to the generic MIPS elements and then creating the precise measurement rules to determine the functional size according to the instruction, called the dictionary.

Rules The first step in our approach is the mapping of MIPS elements to COSMIC elements. The mapping helps identify the meaning of MIPS elements to COSMIC which facilitates the measurement of the functional size while giving room for further improvements. Table 4 shows the proposed mapping rules.

Table 4: MIPS/COSMIC mapping rules

Rule number	COSMIC element	Rule description
1	Functional Process (FP)	Identify 1 functional process for each subroutine in the file.
2	Data movement	Identify 1 Entry (E) for each source register in each instruction in a FP.
3	Data movement	Identify 1 Entry (E) for each immediate each instruction in a FP.
4	Data movement	Identify 1 Exit (X) for a destination register in each instruction in a FP.
5	Data movement	Identify 1 Exit (X) for the new PC value after branch and jump instructions.
6	Data movement	Identify 1 Exit (X) for the return value after branch and link and jump and link instructions.
7	Data movement	Identify 1 Read (R) for each Load instruction inside a FP.
8	Data movement	Identify 1 Write (W) for each Store instruction inside a FP.
9	Functional process size	Aggregate the COSMIC Function Point (CFP) for each data movement in a FP. to obtain the size of the process.

10	Size of the software	Aggregate the CFP of each FP. to obtain the size of the whole software.

Dictionary The dictionary is the actual mapping of the MIPS instruction set to COSMIC. The dictionary consists of 301 entries corresponding to the total number of instructions in the MIPS instruction set (total of 301 instructions). The dictionary is designed to contain information about both MIPS and COSMIC. It has two versions, the detailed main dictionary (for human interaction) and a machine one (for the tool). The detailed dictionary has more details not related to the tool and is divided into eight columns (Opcode, instruction name, Entry, Exit, Read, Write, Total, Exceptions). The machine dictionary has six columns used to calculate the functional size (Opcode, Entry, Exit, Read, Write, Total). The main detailed dictionary is based on understanding the COSMIC rules and having the latest version of the MIPS instruction set manual. The steps behind formatting the dictionary include going through every instruction in the instruction set and understanding the operation that this instruction performs, then mapping the parts of the instruction into COSMIC. To map an instruction from the instruction set manual to the dictionary the following points need to be considered:

- The number of the source registers that will be mapped to Entries.
- The number of the destination registers that will be mapped to Exits.
- Dealing with the memory that will be mapped to Read or Write.
- Adding any exception that might occur.

5 COSMIC Based Automation Prototype for MIPS ISA

In this section the tool is discussed in detail followed by a small-scale validation test.

5.1 The Tool

The tool was created using Eclipse Java. This section covers the logic behind the tool, the user interface and error handling.

The Tool Logic The logical steps implemented in the tool are:

- Save the dictionary in a data structure: the chosen data structure is ArrayList and was chosen for its dynamic data structure and ease of access to any of its elements.
- Once the user has selected the file, the tool reads that file and stores it in an ArrayList.
- The tool takes that ArrayList and performs string manipulation to split each entry and obtain the Opcode.

- The result of the string manipulation may not be an Opcode but a label that identifies the start of a subroutine. The tool keeps track of the labels and its number for the user because the subroutine is mapped into FP.
- The Opcode is then matched with the dictionary entries.
- If the tool finds a matching Opcode, it will increment the total number of instructions, Entries, Exits, Read, and Write.
- The user has the option to save a detailed report after calculating the functional size.

The saved report is named using the following convention: name of the selected file concatenated with the word "Report".
It contains the following information:

1. Total number of lines of code in the file
2. Total number of instructions in the file
3. Total number of subroutines (FP)
4. List of the labels corresponding to each subroutine
5. Total number of entries in the file
6. List of the instructions that cause the aggregate of the Entries count with an indication of the number of Entries corresponding to each instruction
7. Total number of Exits in the file, list of the instructions that cause the aggregate of the Exits count with an indication of the number of Exits corresponding to each instruction
8. Total number of Reads in the file, list of the instructions that cause the aggregate of the Read count with an indication of the number of Reads corresponding to each instruction
9. Total number of Writes in the file
10. List of the instructions that cause the aggregate of the Write count with an indication of the number of Writes corresponding to each instruction
11. The total functional size of the file

Graphical User Interface As shown in Fig. 1, the graphical user interface -(GUI) contains four buttons: BROWSE to choose the file from the computer, CALCULATE to measure the functional size for the selected file, SAVE THE REPORT to save the resulting detailed report as a text file in the same directory as the selected file, and CANCEL to cancel the operation and close the tool.

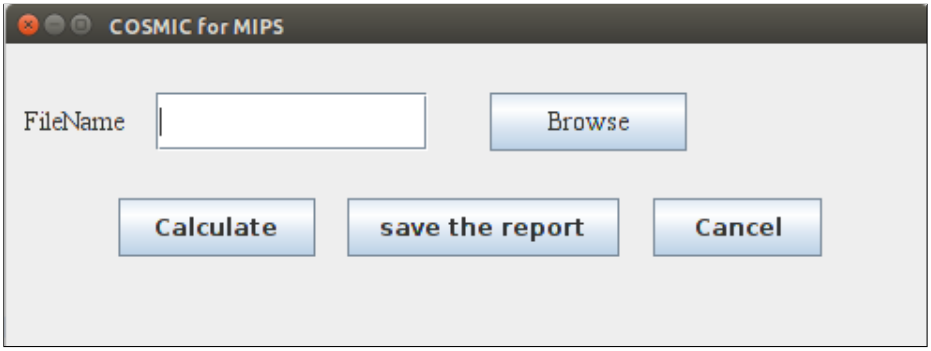

Fig. 1. Graphical User Interface

Error Handling The tool handles some errors such as clicking on the Calculate button without choosing a file 2, clicking on save the report without choosing a file, and clicking on save the report without calculating the functional size.

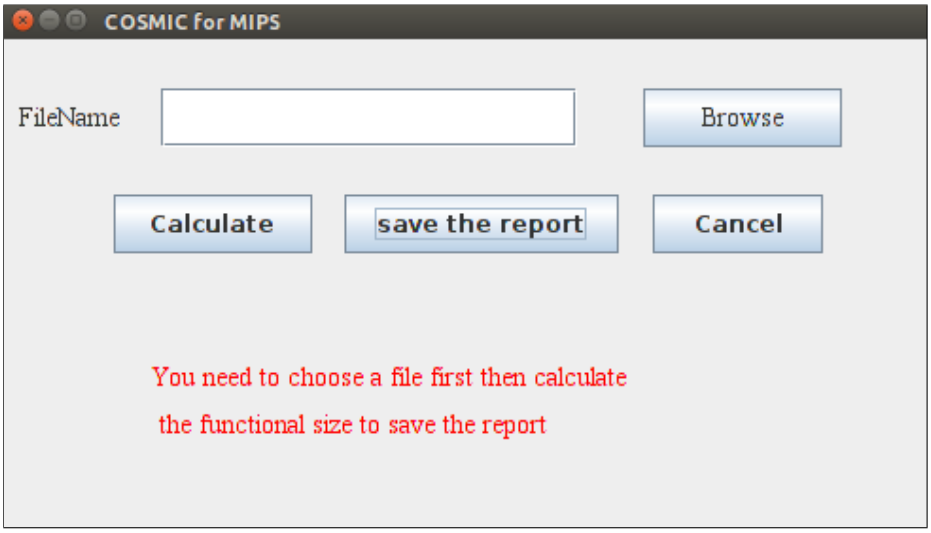

Fig. 2. Clicking on save the report without choosing calculating the functional size.

5.2 Validation of the Prototype Tool

To validate the proposed tool 10 different test files were written. The test files varied in file size, total number of instructions, total number of data movements, and different cases of writing subroutines.
The validation was divided into three phases:

- First phase: read the selected file and compare the Opcode with the dictionary to calculate the functional size. This stage is mainly to check if the tool correctly handles the exceptions that may occur during reading the file, comparison, or even during performing the string manipulation.
- Second phase: write to a file, give the file a specific path, and also attempt to handle any exceptions that may occur.
- Third phase: validate the part responsible for measuring the functional size and the number of subroutines in the selected file.

Test Case Example The file used in our test case is, file written in Assembly based on MIPS, called "TestFile.asm". As shown in Fig. 3, it has a total of 14 lines, 10 instructions, does not have a main label, and has additional two subroutines.

```
    addi $s6 , $0 , 23
    addi $t5 , $0 , 5
    sw $t5 , 0 ( $ s6 )
    addi $s6 , $0 , 8
Loop:
    sll $t1 , $s3 , 2
    add $t1 , $t1 , $s6
    lw $t0 , 0 ( $t1 )
    bne $t0 , $zero , Exit
    addi $s3 , $s3 , 1
    j Loop
Exit:
```

Fig. 3. TestFile.asm

As shown in Fig. 4, the report starts by stating the file name, total number of lines in the file, total number of instructions in the file, and the COSMIC measurement details. The number of functional processes FP identified is 3, namely: 3 main, Loop, Exit. When TestFile.asm does not have a "main" label, the tool automatically adds a label called "main" to the list of labels. The prototype tool identified 22 Entry data movements, nine Exits, one Read and one Write data movements. The total functional size of this TestFile.asm is: 33 CFP. Table 5 shows the result of the COSMIC functional size manual measurement of TestFile.asm.

```
This is the report coresponding to TestFile.asm

The file had 14 lines of code.
The total number of instructions in the file = 10 instruction(s).

COSMIC calculation:

The file had 3 functional processes.
    main
         Loop
         Exit
The functional size = 33 CFP.

The total number of entries = 22.
 The instructions that caused the number of Entries:
             addi $s6 , $0 , 23------> 2 entries.

             addi $t5 , $0 , 5------> 2 entries.

             sw $t5 , 0 ( $ s6 ) ------> 3 entries.

             addi $s6 , $0 , 8------> 2 entries.

             sll $t1 , $s3 , 2 ------> 2 entries.

             add $t1 , $t1 , $s6 ------> 2 entries.

             lw $t0 , 0 ( $t1 ) ------> 2 entries.

             bne $t0 , $zero , Exit------> 4 entries.

             addi $s3 , $s3 , 1------> 2 entries.

             j Loop------> 1 entry.

The total number of exits = 9.
 The instructions that caused the number of Exits:
             addi $s6 , $0 , 23------> 1 exit.

             addi $t5 , $0 , 5------> 1 exit.

             addi $s6 , $0 , 8------> 1 exit.

             sll $t1 , $s3 , 2 ------> 1 exit.

             add $t1 , $t1 , $s6 ------> 1 exit.

             lw $t0 , 0 ( $t1 ) ------> 1 exit.

             bne $t0 , $zero , Exit------> 1 exit.

             addi $s3 , $s3 , 1------> 1 exit.

             j Loop------> 1 exit.

The total number of read = 1.
 The instructions that caused the number of Reads:
             lw $t0 , 0 ( $t1 ) ------> 1 read.

The total number of write = 1.
 The instructions that caused the number of Writes:
             sw $t5 , 0 ( $ s6 ) ------> 1 write.
```

Fig. 4. TestFileReport

Table 5: COSMIC calculation

Rule applied	**Opcode**	**Element**	**Data movement type**	**CFP value**
2	addi	0	E	1
3	addi	23	E	1
4	addi	$s6$	X	1
3	addi	5	E	1
2	addi	0	E	1
4	addi	$t5$	X	1
2	sw	$t5$	E	1
2	sw	$s6$	E	1
3	sw	0	E	1
8	sw	write operation	W	1
2	addi	0	E	1
3	addi	8	E	1
4	addi	$s6$	X	1
2	sll	$s3$	E	1
3	sll	2	E	1
4	sll	$t1$	X	1
2	add	$t1$	E	1
2	add	$s6$	E	1
4	add	$t1$	X	1
2	lw	$t1$	E	1
3	lw	0	E	1
4	lw	$t0$	X	1
7	lw	read operation	R	1
2	bne	$t0$	E	1
2	bne	*zero*	E	1
3	bne	offset (Exit)	E	1
3	bne	pc	E	1
6	bne	new pc value	X	1
2	addi	$s3$	E	1
3	addi	1	E	1
4	addi	$s3$	X	1
3	j	target (Loop)	E	1
6	j	new pc value	X	1
			– Total E: 22 – Total X: 9 – Total R: 1 – Total W: 1	– Total: 33 CFP

6 Conclusion

The goal of this paper was to propose an approach for a 'universal' tool based on COSMIC ISO 19761 to ensure that the measurement of all types of input software written in different programming languages is correctly automated. The 'universal' tool may be achieved by generalizing the approach proposed in this paper to cover, and use, the machine code (ISA-Instruction Set Architecture) generated by any compiler/Assembler to get the COSMIC functional size of a program.

A feasibility prototype tool based on COSMIC and MIPS was developed using Eclipse Java based on the latest version of the MIPS architecture. The tool uses a dictionary to classify the instructions inside a file and gives the user a detailed report of the functional size of the file, including the details of the functional size measurement. This paper was limited to only a specific release of the MIPS architecture and a specific instruction set.

In the future, the accuracy and the precision of the tool will be analyzed with more test cases. The dictionary should be expanded to include all MIPS instructions, including pseudo-instructions and not just those in the ISA. Lastly, generalizing the proposed approach to cover machine code, and developing a plugin version of the proposed tool that can be added to the source-code editors, software development and version control tools, or DevOps lifecycle tools ,such as VSCode, GitHub and GitLab, to automate the measuring process of the COSMIC functional size in each run or deployment.

References

1. Common Software Measurement International Consortium (COSMIC): Measurement Manual v4.0.1 (2015).
2. HOPL homepage, http://hopl.info/
3. Prechelt, Lutz. An empirical comparison of seven programming languages. Computer 33.10 (2000): 23-29.
4. H. Soubra, A. Abran, S. Stern and A. Ramdan-Cherif. Design of a Functional Size Measurement Procedure for Real-Time Embedded Software Requirements Expressed using the Simulink Model. Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement, Nara, 2011.
5. Colin Hammond, Erdir Ungan and Alain Abran. Automated COSMIC Measurement and Requirement Quality Improvement Through ScopeMaster Tool. Joint International Software Measurement Workshop and MENSURA International Conference – IWSM-MENSURA, Beijing, China, September 2018.
6. Scopemaster website, https://www.scopemaster.com/
7. Lamma, Mello and Riguzzi, A system for measuring Function Points from an ER-DFD specification, The Computer Journal, pp. 358-372, 2004.
8. K. Lind and R. Heldal, A Model-Based and Automated Approach to Size Estimation of Embedded Software Components, in ACM/IEEE the 14th International Conference on Model Driven Engineering Languages and Systems, Wellington, New Zealand, 2011.

9. S. Azzouz and A. Abran, A proposed measurement role in the rational unified process and its implementation with ISO 19761: COSMIC-FFP, in Software Measurement European Forum, Rome, Italy, 2004.
10. H. Diab, F. Koukane, M. Frappier and R. St-Denis, μ c ROSE: automated measurement of COSMIC-FFP for Rational Rose RealTime, Information and Software Technology, vol. 47, no. 3, pp. 151-166, 2005.
11. R. Gonultas and A. Tarhan, Run-Time Calculation of COSMIC Functional Size via Automatic Installment of Measurement Code into Java Business Applications, in IEEE 41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2015.
12. H. Soubra, L. Jacot and S. Lemaire, Manual and Automated Functional Size Measurement of an Aerospace Real-time Embedded System: A Case Study based on SCADE and on COSMIC ISO 19761, 2015.
13. K. Lind and R. Heldal, A Model-Based and Automated Approach to Size Estimation of Embedded Software Components. MODELS'11 - 14th International Conference on Model Driven Engineering Languages and Systems, pp. 334-348.
14. MIPS website, https://www.mips.com/