Functional Complexity Measurement

De Tran-Cao de.trancao@lrgl.uqam.ca
Alain Abran abran.alain@uqam.ca
Ghislain Lévesque levesque.ghislain@uqam.ca
Software Engineering Management Research Laboratory
University of Quebec at Montreal (UQAM)

Abstract

The specific analysis of FPA, from a complexity viewpoint, leads us to propose an initial model of functional complexity in which software complexity is a function of component complexity and system complexity.

In this paper, we will use the next generation of functional size methods proposed by the COSMIC team [1][24], and we will look at it from the complexity perspective to identify some factors that affect complexity. Based on the analysis of such factors, we will propose a model for measuring a specific perspective of software complexity, which we will refer to as functional complexity.

This model of functional complexity has two parts: component complexity, that is, the complexity of a functional process (COSMIC terminology) that comes from both the data movements and data manipulation; system complexity, that is, the complexity coming from relationships between the functional processes like communication, concurrence and multi-instances. Measuring these factors independently gives us a set of indicators or baselines for assessing software complexity from a functional perspective.

Such a measure of functional complexity will then be used in the future in empirical studies to investigate its contribution to the improvements of estimation models which sometimes fare poorly when based only on functional size.

1. Introduction

Software size is used as a key factor in the evaluation of development effort and productivity. Two approaches widely known in the literature for estimating software size are a posterior estimation such as line of code (LOC) [10] and a priori estimation such as function point analysis (FPA) [7]. FPA quantifies software size in terms of function points which can be determined from software artifacts like the requirement specifications, design specifications, etc. Therefore, FPA is more useful than LOC for predicting early development effort [6][15].

However, FPA has also been greatly criticized. FPA is not widely accepted for measuring the size of some types of software, such as real-time software and scientific software, due to some weaknesses such as not taking into account the complexity of algorithms and various other characteristics of real-time software [3][21][26]. It has also been criticized for the weights of the functional types and the degrees of influence of the 14 general system characteristics (GSCs) that have been determined subjectively both in terms of their calibration scales and in the specific selection of weights and degrees of influence [4][5][11][23].

The field of research on software complexity investigates the assumption that the complexity of software is an important indicator for estimating software development effort. Some researchers even postulate that complexity, and not size, may be the most relevant characteristic in estimating effort [22]. From the perspective of complexity, FPA measures the size of software by quantifying the complexity of some *base functions* (Input, Output, Inquiry, etc.), as well as of some *system complexity factors*. This leads implicitly to two categories of software complexity: component complexity and system complexity. Component complexity comes from the components of the software, and system complexity comes from the general characteristics of the system. Other researchers agree that there are two parts to software complexity: the complexity of the components and the complexity of the system [9].

In this paper, we use the generic model of software proposed by the COSMIC team [1][2][24] to develop a design for a measure of software functional complexity. This COSMIC model addresses the component view of the software functional process. On the one hand, component complexity is defined as the "internal complexity" of functional processes. System complexity, on the other hand, is defined as the complexity in the relationships between the functional processes. We propose some factors and simple measures of these factors for quantifying functional complexity. The proposed measures of these factors will provide baselines for an evaluation of software complexity. They might also be used for many other purposes, for example, choosing an environment or a language for developing software, explaining the difficulty of different tasks, etc.

2. What is complexity?

The first challenge when talking about measuring software complexity is to answer the question: "What is complexity?"

IEEE defines software complexity as "the degree to which a system or component has a design or implementation that is difficult to understand and verify" [14].

Basili defines complexity as a measure of the resources expended by a system while interacting with a piece of software to perform a given task. If the interacting system is a programmer, then complexity is defined by the difficulty of performing tasks such as coding, debugging, testing or modifying the software [8].

There is no consensus on how to define software complexity, and Zuse says that the term complexity measure is a misnomer: The true meaning of the term software complexity is the difficulty to maintain, change and understand software [25].

These definitions associate software complexity with the difficulty of performing a task on the software. An implicit assumption is that software complexity correlates well with the work effort (man-hours) required to develop or maintain the software. Among the best known attempts at measuring software complexity are: Software Science [12] which deals with the difficulty of code comprehension, the Cyclomatic Number [20] which deals with the structure of code, and Information Flow [13] which deals with the relationship of modules. More recently, six metrics have been proposed to measure some baselines in terms of Object Oriented Design, like Number of Class, Number of Children, Depth of Inheritance Tree, etc. [18].

The term 'functional complexity' in this paper is interpreted as a candidate explanatory variable for investigating the work effort required to develop the software function, including decomposing and allocating the functional processes and designing each functional process to fulfill user needs as stated in the software specifications.

3. Software model and software complexity model

In the family of measurement methods based on software specifications, only COSMIC-FFP [1][2][24] explicitly proposes a generic software model (Figure 1) which is based on functional requirements.

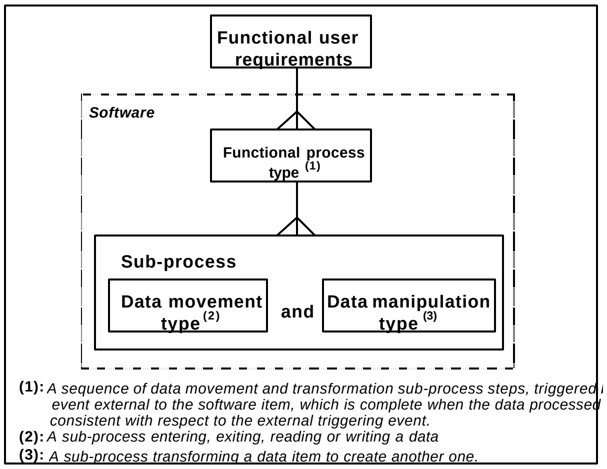
Functional user requirements
Software
Functional process type (1)
Sub-process
Data movement type (2)
and
Data manipulation type (3)
(1): A sequence of data movement and transformation sub-process steps, triggered
event external to the software item, which is complete when the data processed
consistent with respect to the external triggering event.
(2): A sub-process entering, exiting, reading or writing a data
(3): A sub-process transforming a data item to create another one.

Figure 1: COSMIC-FFP generic software model [1]

According to this model, even at the earliest stages of the software life cycle, software is considered as a set of functional user requirements (FURs) that are implemented by a set of functional processes. Each of these is an ordered set of sub-processes for fulfilling the functional process. There are two types of sub-process: *data movement type* and *data manipulation type*.

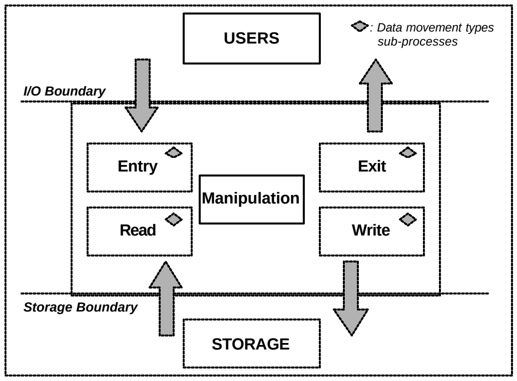

Figure 2: COSMIC-FFP sub-process types [1]

The COSMIC team has proposed a measurement method called COSMIC-FFP, in which all data movements are captured and taken into account (Figure 2). The method does not deal with data manipulation for functional size purposes. However, from the COSMIC point of view, a functional process is activated by one triggering event. When it is triggered, it receives the input data (Entry, Read), manipulates data and generates the output data (Write, Exit).

Here, COSMIC-FFP is not analyzed from its initial perspective, that of functional size. Instead, we analyze it from a complexity perspective, that is, in terms of component complexity and system complexity. Of course, COSMIC-FFP addresses only a part of component complexity because it takes into account only the number of data movements; it deals neither with data manipulation nor characteristics of the whole system. Can only data movements indicate the work-effort required to analyze, design and code a functional process? Is there any complexity coming from the relationship between the functional processes?

We begin with some observations on the generic software model in Figure 1.

3.1. Two types of complexity

From the user's point of view, software is like an integrated set of programs, documents and data for resolving a problem or a set of problems. To the user, software can be considered as a set of functions. Indeed, the software specifications must describe all the features of the software among which is the set of FURs. This set describes all the functionalities that the software must perform, but ignores the question of how to do so. The software product and the set of FURs are the same when comparing software functions, but are, of course, expressed differently. The FURs will be implemented by a set of functional processes. There are, of course, *difficulties in decomposing or allocating FURs in the functional processes and in designing each process to meet the user needs*. The former means the complexity in the relationships between the functional processes and the latter means the complexity within each functional process.

3.2. Complexity in relationships between the functional processes

The allocation of FURs is the decomposition of FURs into a set of coherent functional processes. This task determines the function of each of the processes, and the relationships between them, designed to meet the FURs. Intuitively, software is not a set of independent functional processes. In fact, many functional processes in the system must be well coordinated to fulfill the user needs. We identify three primary types of relationships between functional processes:

- **Control and data communication**: This relationship addresses the communication between two processes. Two processes may be located on the same site or on two different sites. A process may send a triggering event (with or without accompanying data) to activate another process. The former may receive data (feedback) from the latter (Figure 3). The complexity here is the difficulty in determining the function of each functional process and the way in which they cooperate to fulfill the user needs. It represents the coupling of two processes. The term coupling was proposed by Yourdon and Constantine [11], and refers to the degree of interdependence of modules. In the glossary of COSMIC-FFP [1], coupling is defined as a measure of interconnection among functional processes. Coupling depends on the complexity of the interface between functional processes, the point at which entry or reference is made to a functional process and what data crosses the interface. Roughly speaking, this term is used to indicate the degree of interdependence of two processes. In fact, two processes having control and/or data communication may be coupled in many ways, for example: in a communication protocol or in parallel, or they may be synchronized. They are also coupled from a functional point of view because each of them carries out a part of the work (computational task or functional user requirement). Control and data communication could also be associated with some of the 14 GSCs of FPA, like data communication or distributed data processing, on-line data entry and a part of complex processing [6]. It could also be associated with parallelism and synchronization in real-time systems [21].

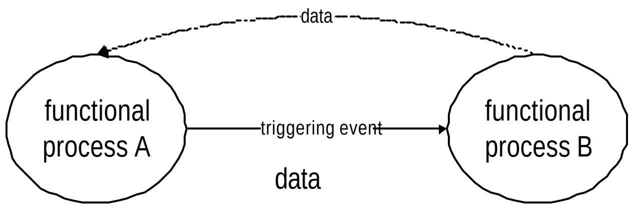

Figure 3: Communication between two processes.

- **Concurrence**: there is concurrency when more than one process works in a mutually exclusive mode but they are triggered simultaneously. Each group of processes in concurrence needs a special process to control it or at least a mechanism to deal with it. This characteristic is very common in multi-task, multi-user environments and in real-time systems. This increases the difficulty in designing, coding and testing the software. Many kinds of concurrency can be observed in the sharing of resources such as time, processor, memory, etc. However, whereas they are often identified in the system design, it is not easy to identify them from the specifications. In the specifications, concurrency in the sharing of process data can sometimes be identified (Figure 4): where two processes simultaneously access one data group in the mutually exclusive mode, there is concurrency in accessing data.

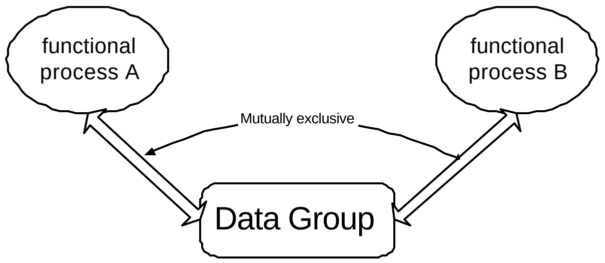

Figure 4: Concurrence in accessing data

This relationship could be associated with the on-line update GSCs of the FPA method [6] and to the concurrency characteristic in Assert-R [21].

- **Event or instance handling**: from the point of view of COSMIC-FFP, a process is triggered by an event. COSMIC-FFP does not deal with how often the process is triggered. Since the triggering event of a process may come from many different sources, the process may have many instances associated with each event, and the system must at least have a mechanism to handle these events (Figure 5).

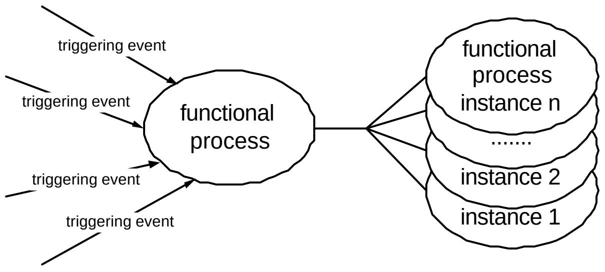

Figure 5: Many instances of a process.

For example, the process for withdrawing money automatically from a bank machine can be triggered by many people simultaneously at different machines. If the system is distributed or multi-tasked, we can find many instances of the process. Each instance deals with an event. If the system is consequent, then there is at least a mechanism (queuing, for example) to handle these events. This characteristic can be associated with the data transaction rate of FPA and the assumption is that *more effort is needed to analyze and design a mechanism for managing these instances, or for handling the events*. This characteristic is also very important in real-time systems because the event must be responded to within specified time constraints.

In our experience, these relationships are fairly common, especially in real-time software, and they increase software functional complexity in terms of the difficulty in analyzing, designing and implementing solutions. Therefore, they are worth taking into account in measuring software complexity.

3.3. Complexity in each functional process

The complexity in each functional process can be associated with the difficulty related to developing it. Generically, a process is a black box which receives input data, manipulates them and produces output data. Designing a process is determining a set of tasks or sub-processes which must be executed to complete the process, and includes some main aspects of the way to manipulate the input data to produce the expected output. COSMIC-FFP proposes a process model which is a set of data movements and data manipulations. On the one hand, in measuring functional size, the COSMIC-FFP size model does not take into account the data manipulations. On the other hand, data manipulation is a sub-process transforming one data item into another one; it can be interpreted as related to algorithm complexity [16] or to a part of processing complexity [6]. It is worth noting that often little is known from the specifications document about how the algorithm manipulates input to produce output. Some aspects of algorithms can be:

- **Different cases** determined by the value of input – output from the specifications: Normally, the specifications describe not only the input and output of the process, but also the conditions on inputs to produce different expected outputs. For example, the specification for the process that verifies the user's identification may be the following: The user enters his user name and password from the console. The process tests whether or not the user name is correct. If the user name is incorrect, the user is asked for another user name with an error message "User name is incorrect." If it is correct, the process verifies the password. If the password is correct, the main menu is triggered to help the user perform the operations. If not, a message "password is incorrect" is shown and the user is asked for another password. If the user fails the verification process three times, the account is locked. In this case, we have four different outputs depending on the input values: message “Password is incorrect”, message “User name is incorrect”, triggering event for the main menu, and triggering event for locking the user account. Consequently, the coding effort (and the number of lines of code) depends on the number of these cases. If only the input and output are taken into account, we have the same value for a process with a few cases as we do for one with many more cases. Therefore, for complexity measurement, a measure is proposed based on the distinct number of cases, rather than on input and output. Each case may be represented as a decision rule:

IF (condition on input values) THEN (expected outputs)

In addition, from the specifications, many constraints can be specified. There are two types of constraints: functional constraints like the integrity constraints and business rules, and non-functional constraints like time constraints, constraints on the development process, constraints on standards, etc. A functional constraint can also be interpreted as a case that is represented as a decision rule.

Moreover, each condition on inputs can be considered to represent one state of a functional process. A functional process may exhibit different behaviors in reaction to one event. Its behavior depends on its state at the time of the event. 3D Function Points [26] suggests that states and transitions are the primary contributors to complexity. But the difficulty is that the state diagram is not always available in the documented software artifacts. Therefore, we must often measure state and transition via different cases of functional processes.

- **Data movements**: Another aspect of the complexity of a process is the number of data movements that must be performed in the process. A data movement may receive data from the user side (Entry) or from the storage device (Read), move data to the storage device (Write) and produce data output to the user side (Exit). Many methods with a functional approach (FPA, Mark-II, 3D Function Points, COSMIC-FFP, etc.) use the data movements in the process as a key factor contributing to the functional size of the software. The names, definitions and measures of data movements may vary from method to method, but in general they take into account the quantity of data input-output as a representative indicator of functional size. We therefore propose, similarly, that data movements be considered as a factor of complexity since they can be intuitively associated with the tasks that must be performed in the process.

At the specification stage, sometimes little is known about the specifics of an algorithm. What can be most easily observed, however, about the algorithm for implementing the process are the *different cases of the process and data movements in the process*. These may be interpreted as the number of logical steps which the algorithm must take into account. Hence, the measurement of these two factors is proposed as two indicators of component complexity, i.e. the complexity of functional processes.

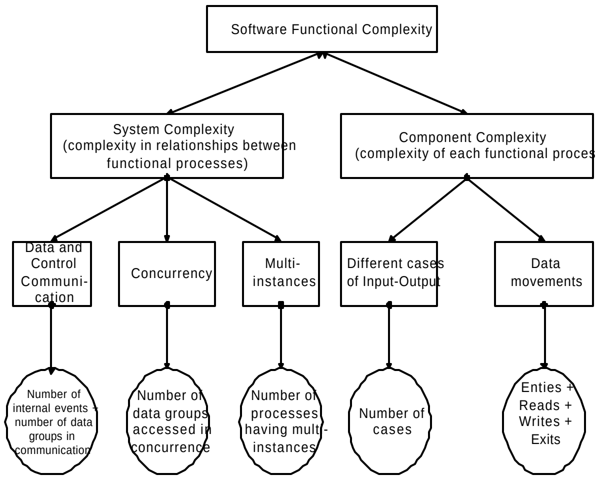
Software Functional Complexity
System Complexity (complexity in relationships between functional processes)
Component Complexity (complexity of each functional proces
Data and Control Communi-cation
Concurrency
Multi-instances
Different cases of Input-Output
Data movements
Number of internal events + number of data groups in communication
Number of data groups accessed in concurrence
Number of processes having multi-instances
Number of cases
Enties + Reads + Writes + Exits

Figure 6: Software functional complexity model

From the analysis above, a generic model for software functional complexity can be derived, as illustrated in Figure 6, in which software complexity is viewed as the complexity of components and the complexity of the system itself combined. The complexity of components is related to the *different cases of the process* and to the *data movements* occurring in the process. The complexity of the system is related to the complexity coming from the relationships between the functional processes of the system. This complexity can be seen as the work effort required to decompose the FURs into a set of coherent functional processes to fulfill the FURs. It is characterized by *control and data communication*, *concurrency* between the processes and *event or instance handling*.

4. Measurement methods

Some simple measures are now proposed for assigning numerical values to the complexity factors proposed above. These measures will provide indicators of software functional complexity, defined as a function of five measures of the five factors in Figure 6. The following definitions are, however, necessary before proposing specific measures for these factors.

To measure software complexity, we must know whether something belongs to the software or not. The boundary of software (or a piece of software that we want to measure) is the conceptual frontier between the software and the environment in which it operates. The boundary distinguishes what belongs to the software and what does not. In terms of functional processes, we need to determine which belong to the software. These functional processes may be located on different sites. Moreover, a functional process may be installed on many sites. We do not consider a process installed on many different sites as a different process, but something going out from it may increase the complexity of the system. In this perspective, attention is paid to the triggering events in the system, rather than to the number of processes installed for one logical process.

A functional process, in the COSMIC-FFP measurement method, is an ordered set of data movements (Entry, Exit, Read, Write) implementing a cohesive set of FURs. It is triggered by an event and, once performed, must leave the software in a coherent state with respect to the triggering event. A data movement is a logical task perceived from the user's point of view; it is a movement of a data group in or out of the process. A data group is known as a set of attributes describing something in the real world (or in the problem domain). A data movement relates to only one data group.

A triggering event is an event such that, when it occurs, the software must do something to respond to it. A triggering event may occur either outside the boundary of the software or inside it. For example, a timing event is a type of event coming from outside the software (from a timing device); while an event like "failure to give the correct password three times" occurs inside the software boundary that is generated by a process of the software. So, two types of triggering events are distinguished: external events and internal events, according to where they come from, outside or inside the boundary. A triggering event must be identified in the software specifications. That means it must be mentioned in the problem domain. One simple measure for each of the five complexity factors mentioned in Figure 6 are now proposed.

Control and Data Communication

The communication between two processes is a concept describing the situation in which a process sends out a triggering event to activate another process. The latter may or may not receive data from the former, and may or may not send the result back to the former.

For the two communicating processes, a simple measure is proposed to quantify their complexity: the number of data groups in communication between them plus one (+ 1 to take into account the triggering event). Therefore, the complexity of control and data communication (CDC) of the software is:

$$CDC = \sum(\text{number of data groups in communication between two processes} + 1)$$

The number of data groups communicating between members of all process pairs are summed in this way. Thus, the complexity of control and data communications is defined as the summation of all internal triggering events and all data groups used for communications occurring in the system.

Concurrency

Concurrency is a concept describing the situation in which more than one process may access one data group simultaneously when they work, in principle, in a mutually exclusive mode. Logically, each data group accessed concurrently by many processes needs a mechanism or a special process (monitor) to control the concurrency. The principle for resolving the concurrency may be the same for many different data groups. But, for each data group in question, there is a need to analyze and design a mechanism like this; we therefore propose to select the number of data groups accessed concurrently as the indicator of the complexity related to the concurrency between processes. We denote this complexity as concurrency complexity (CC).

CC = number of data groups accessed concurrently in the system

Multi-instance Handling

A functional process may be triggered by many different triggering events. It can also be triggered many times by an event at different moments. When the functional process is working to respond to an event, another event may occur and request the response of the process (interrupt). This case at least needs a mechanism to handle the events or create many instances of a process to deal with all events. For simplicity sake, we use the term multi-instance for both handling events and creating many instances of a process. For the purpose of measuring this characteristic, we take into account one for each process having many instances. The number of these processes indicates the effort to analyze and design a mechanism to handle events in the system and then to implement it. Therefore, *how many processes have events that need handling is an indication of how much* *effort is needed to deal with them*. We denote this complexity as multi-instance complexity (MIC):

MIC = number of processes having multi-instances

The three measures proposed above can be used to provide an indication of the difficulty in developing a system, rather than how large the system is (how many functions it has). These measures can be used to explain why real-time or engineering software is perceived as being "more complex" or "more difficult" or needing much more effort" than MIS software with the same size in term of functions.

Different cases of functional process

As already mentioned, a functional process may have many different cases represented by different input-output pairs. Each case can be interpreted as a decision rule:

IF (condition on inputs) THEN (desired outputs)

These cases give an outline concerning the tasks that must be designed and fulfilled by the algorithm used to implement the process. So, the number of different cases is proposed as an indicator of algorithmic complexity. This number may tell us how many tasks must be dealt with in the algorithm, the assumption being that there is a relationship between the number of tasks and how much effort is required to handle them. This complexity is referred to as different case complexity (DCC).

DCC = number of different cases of the process

= number of

IF (condition on inputs) THEN (desired outputs)

Data movement in the process

A data movement is a logical task in the process. Generically, a process receives input data, manipulates them and produces output data. Some logical tasks may be identifiable and intelligible from the user's viewpoint (from the specifications or interface design), such as: Entry, Read, Write, Exit (proposed by COSMIC-FFP). The number of these tasks has been defined by consensus within the COSMIC group to represent the "functional size" of a process. Intuitively, it indicates how many tasks the process must carry out. The number of data movements is identified and taken into account in the same way that COSMIC-FFP does so. Four types of data movements are identified and measured, using the definitions and measurement rules proposed by COSMIC-FFP. This, from a functional complexity viewpoint, is now defined here as data movement complexity (DMC)

DMC = number of Entries + number of Exits + number of Reads + number of Writes

Aggregate results and estimate of software complexity

In summary, the measures of each of five different factors of software functional complexity are: control and data communication, concurrency, multi-instance handling, different cases of process and data movements in the process. They are defined as the indicators of the effort required in different tasks in software development. They can, of course, be used as independent measures to describe or quantify the different aspects of software: for example, CDC may be used to compare the complexity in data communication of two software products. They can also be used for choosing the environment or programming language; for example, if software is "communication strong", we might choose an environment in which communication programming is well supported.

This paper also proposes that the software functional complexity (SFC) is a mathematical function of these five measures.

SFC = f(CDC, CC, MIC, DCC, DMC)

The investigative work to establish such a function, f, still has to be performed and will require much more empirical research.

5. Conclusion

In this paper, software functional complexity was considered from a generic perspective based on lessons learned from an idea of Card and Glass [9], from the FPA method [6][7] and from the COSMIC-FFP method [1][3]. Two categories of software functional complexity are proposed: complexity of components and complexity of the system. The complexity of components is derived from the components of software in the software model as modules or functions. The complexity of the system is derived from relationships between the components or characteristics of the software. The software functional model proposed by the COSMIC team was used as the basis for studying software functional complexity within these two categories.

Three factors of system complexity were proposed, each representing a relationship between the functional processes of software: control and data communication, concurrence and multi-instance handling. These factors are generally, and strongly, related to some of the 14 GSCs proposed by FPA and some complexity factors mentioned in Asset-R and 3D

Function Points, like concurrency and synchronization. They also are typical characteristics in real-time systems and engineering systems.

To quantify component complexity, i.e. the comp lexity in each functional process, two factors were examined: the different cases of process and the data movements in the process. We believe that the complexity of the algorithm used to implement a functional process is worth studying, but, at the specifications stage, little is known about how the process manipulates data input to produce the desired output. What can be learned about the algorithm in the early phases, such as analysis and design, are the logical tasks in the process like data movements (Entry, Read, Write, Exit) and the different cases dependent on the specification of input values and desired output pairs. These two factors are, therefore, to be quantified and used as indicators of the complexity of functional processes.

In this work, a simple measure has been proposed for each of the above factors. These measures may be used independently for different purposes, and also as parameters for assessing software functional complexity. However, the main purpose of this work was to establish and build progressively on a software functional complexity model rather than a complete measuring method of the still ill-defined global concept of software complexity. In the future, empirical research will be initiated to investigate the relationships bet ween these factors to derive a unique measure for software functional complexity.

6. References

[1] Abran, A.; Desharnais, J.-M.; Oligny, S.; St-Pierre, D.; Symons, C., *COSMIC FFP - Measurement Manual – Field Trials Version*, Montreal, May, 2001.

[2] Abran, A., St-Pierre, D., Oligny, S., *Improving Software Functional Size Measurement*, LRGL, UQAM, 1999.

[3] Abran A., Desharnais J.-M., Maya M., St-Pierre D., Bourque P., *Design of a functional size measurement for Real-Time Software*, research report No 13-23, 1998.

[4] Abran A., Jacquet J-P, *Metric Validation proposals: A Structured Analysis, Presented at the 8th International Workshop on Software Measurement,* Magdeburg, Germany, September, 1998.

[5] Abran, A, *Analyse du processus de mesure des points de fonction*, in Département de génie électrique et de génie informatique. Montréal: École Polytechnique de Montréal, 1994, pp. 405.

[6] Albrecht A., Gaffney J., Software Function, Sources Lines of Code, and Development Effort Prediction: A Software Science Validation, *IEEE Transactions on Software Engineering*, Vol. SE-9, no 6, November 1983.

[7] Albrecht A., *Measuring Application Development Productivity*, Present at IBM Applications Development Symposium, 1979.

[8] Basili V.R., *Qualitative software complexity models: A summary in tutorial on Models and methods for software Management and Engineering*. IEEE computer Society Press, Los Alamitos, California, 1980.

[9] Card D.N., Glass R.L., *Measuring Software Design Quality*, Prentice Hall, 1990.

[10] Conte, S.D., Dunsmore, H.E., and Shen, V.Y., *Software Engineering Metrics and Models*, Benjamin/Cummings, Menlo Park, CA, 1986.

[11] Fenton N.E., Pfleeger S.L., *Software Metrics: A Rigorous and Practical Approach*, Second edition, 1997.

[12] Halstead, M.H., *Element of Software Science*, New York, Elsevier North-Holland, 1977.

[13] Henry S.M., Kafura D.G., Software Structure Metric Based on Information Flow, *IEEE Transactions on Software Engineering*, Vol. 7, no 5, September 1981, pp.545-522.

[14] IEEE Computer Society: *IEEE Standard Glossary of Software Engineering Terminology*, IEEE std. 610.12-1990, IEEE.

[15] Jeffery R.D., Low C.G.; Function Points in the Estimation and Evaluation of Software Process, *IEEE*, Vol. 16, no 1, January, 1990.

[16] Jones C., *Applied software measurement – Assuring productivity and quality***,** New York, McGraw-Hill Inc, 1991.

[17] Kearney, K.J., Sedmeyer R.L, Thompson W.B, Gray M.A., Adler M.A., Software Complexity Measurement, *Communications of the ACM*, Vol. 29, no 11, November 1986.

[18] Kemerer, F.C., Chidamber R.S., A Metric Suite for Object Oriented Design, *IEEE transactions on Software Engineering*, Vol. 20, no 6, June 1994.

[19] McCabe T.J., Butler C.W., Design complexity measurement and testing, *Communications of the ACM*, Vol. 32, no 12, 1989.

[20] McCabe, T.J., A Complexity Measure, *IEEE Transactions of Software Engineering*, Vol. SE-2, no 4, p. 308-320, December 1976.

[21] Reifer D.J., Assert-R: A Function Point sizing Tool for scientific and Real-Time Systems, *Journal of Systems Software*, Vol. 11, p. 159-171, 1990.

[22] Sellers, H., *Object-Oriented Metrics – Measures of Complexity*, Prentice Hall, New Jersey, 1996.

[23] Symons, C., Grant Rule P., *One size fits all 'COSMIC' – Aims, Design principles and Progress*, Project Control for Software Quality, ISBN 90-423-0075-2, 1999.

[24] Symons, C., Function Point Analysis: Difficulties and Improvements, *IEEE Transactions on Software Engineering*, Vol. 14, no 1, January, 1988.

[25] Zuse H., *Software Complexity Measures and Methods*, Walter de Gruyter, Berlin – New York, 1991.

[26] Whitmire, S.A., *3D Function Points: Scientific and Real-Time Extensions to Function Points*, presented at Pacific Northwest Software Quality Conference, 1992.