The COSMIC Functional Size Measurement Method

Version 4.0.2

Measurement Manual

(The COSMIC Implementation Guide for ISO/IEC 19761: 2017)

Version 4.0.2 October 2017

Acknowledgements

The following table gives the history of the versions of this document

DATE	REVIEWER(S)	Modifications / Additions
1999-03-31	Serge Oligny	First draft, issued for comments to reviewers.
1999-10-29	COSMIC Core team	Revised, final comments before publishing 'field trial' version 2.0.
2001-05-01	COSMIC Core Team	Revised for conformity to ISO/IEC 14143-1: 1998 + clarifications on measurement rules to version 2.1.
2003-01-31	COSMIC Measurement Practices Committee	Revised for conformity to ISO/IEC FDIS 19761: 2002 plus further clarifications on measurement rules to version 2.2.
2007-09-01	COSMIC Measurement Practices Committee	Revised for further clarifications and additions to the measurement rules to version 3.0, particularly in the area of the Measurement Strategy phase. The method name was changed from the 'COSMIC-FFP method' to the 'COSMIC method'. In upgrading to v3.0 from v2.2, parts of the 'Measurement Manual' v2.2 were separated into other documents.
2009-05-01	COSMIC Measurement Practices Committee	Version 3.0 revised to v3.0.1 to make minor editorial improvements and clarifications, and to distinguish examples more clearly. This version also incorporates the changes proposed in Method Update Bulletins 3, 4 and 5.
2014-04-01	COSMIC Measurement Practices Committee et al	Version 4.0 revised to take into account of Method Update Bulletins 6 to 11, several editorial improvements and to incorporate the Glossary of terms. See Appendix E for details of these changes.
2015-04-01	COSMIC Measurement Practices Committee et al	Version 4.0.1 revised to take into account an error in v4.0 and several other editorial improvements. See Appendix E for details of these changes.
October 2017	COSMIC Measurement Practices Committee	Version 4.0.2 revised to take into account MUBs 12, 13 and 14 and several editorial improvements. See Appendix E for details of these changes.

Foreword

The COSMIC method provides a standardized method of measuring a functional size of software from the domains commonly referred to as 'business application' (or 'MIS') software, 'real-time' software, 'infrastructure' software and some types of scientific/engineering software.

The COSMIC method was originally accepted by ISO/IEC JTC1 SC7 as an International Standard in December 2002. The current version is ISO/IEC 19761:2017 'Software Engineering – COSMIC – A functional size measurement method' [1] (hereafter referred to as 'ISO/IEC 19761').

The ISO/IEC 19761 standard contains only the fundamental normative definitions and rules of the method as at version 4.0 of the method. The purpose of the Measurement Manual is to provide these rules and definitions, and also to provide further explanation and many more examples in order to help Measurers to fully understand how to apply the method. The Measurement Manual is the principal standard description of the COSMIC method for practical use.

Introduction to the COSMIC method

In addition to the 'Measurement Manual', the 'Introduction to the COSMIC method of measuring software' [2] gives a summary of the method. This 'Introduction' document should be read first by anyone who is new to functional size measurement ('FSM') or who is familiar with another FSM method and is thinking of converting, or who simply wants an overview of the COSMIC method, before reading this Measurement Manual. Much background information on FSM and the COSMIC method, as well as supporting Guidelines (e.g. on how to apply the method in various special circumstances, etc.), case studies, research papers, etc., can be found at the portal of www.cosmic-sizing.org .

Main changes for version 4.0.2 of the COSMIC Method

This v4.0.2 introduces no changes in the method's basic principles. The only changes affect definitions and rules and are all designed to improve ease of understanding and measurement repeatability. The most important changes occur in sections 3.2 and 3.3, and originate from Method Update Bulletins (MUBs) which have already been published, as follows.

- The definition of 'Triggering event' has been modified to make clear that only the first data group generated by any one functional user will be moved by a triggering Entry. The possibility of a 'dual role' of a functional user (it may also be an object of interest) is also made clear in the definition of a functional user (MUB 12).

- A new rule to 'identify different data groups (and hence different objects of interest) moved in the same one functional process' has been added (MUB 13)[1].

- In the definition of 'object of interest' the phrase 'process and/or move data' has been replaced by 'move a data group in or out of the software, or to or from persistent storage'. This change aims to make clear that an object of interest may be identified only if it is the subject of a data group that is moved. An object of interest may not be identified as a subject of only data manipulation. (MUB 14).

See Appendix E for a detailed list of all changes made from version 4.0 to v4.0.1 and from v4.0.1 to v4.0.2. This Measurement Manual for version 4.0.2 of the method becomes the current standard definition of the method from its date of publication.

---

[1] These new rules were first published in the Guideline for sizing Business Applicaiton Software, version 1.3, published in May 2017.

Consequences of the main changes to v4.0.2 on existing size measurements, etc.

The original basic principles of the COSMIC method have remained unchanged since they were first published in the first draft of the Measurement Manual in 1999. This is in spite of the various refinements and additions needed to produce the International Standard and to produce all versions of the method up to this latest version 4.0.2.

Functional sizes measured according to the principles and rules of version 4.0.2 of the Measurement Manual may differ from sizes measured using earlier versions only because the new rules intend to be more precise and complete. Hence Measurers have less discretion for personal interpretation of the rules than was possible with earlier versions.

As a further indication of the continuity of the method, anyone who passed the Foundation level certification examination for version 3.0/3.0.1/4.0 of the method will be considered to be still certified for v4.0.1 and v4.0.2 of the method at the Foundation level.

Note on terminology

For the terminology used in this Measurement Manual, see the Glossary in Appendix F. This Measurement Manual uses standard ISO terminology, namely

- 'shall' indicates a rule is mandatory; 'should' indicates a rule is advisory. (If neither term is present assume 'shall''.)
- 'may' indicates 'is allowed to'; 'can' indicates 'is able to'

The contents of this Guideline

Chapter 1 treats the types of software for which the COSMIC method may be used. The term 'Functional User Requirements' ('FUR'), is defined, together with the basic principles of the COSMIC method. The COSMIC method measurement process and the unit of measurement are also defined.

Chapter 2 describes the 'Measurement Strategy', the first phase of the process to measure a piece of software, in terms of its key parameters such as the purpose of the measurement, the scope of the measurement and the functional users of the software. These parameters must be defined before starting to measure so that the meaning of the resulting measurements can be agreed and understood.

Chapter 3 discusses the second 'Mapping' phase of the measurement process by defining how the FUR must be mapped onto functional processes and their data movements. The output of this phase will be the COSMIC 'model' of the FUR that can be measured.

Chapter 4 describes the final 'Measurement' phase of the measurement process. It defines rules for assigning a size to the FUR of a piece of software and how to aggregate sizes of different pieces of software. This chapter also treats how to size changes to software and discusses the possibility of 'local extensions' to the standard COSMIC method.

Chapter 5 lists the parameters that should be considered for recording the measurement.

The Appendices A to E provide some more detail, a summary of the method' principles and rules, and the main changes to this Measurement Manual from 4.0 to v4.0.1 and to the present 4.0.2. Appendix F contains the Glossary of Terms of the method.

The Common Software Measurement International Consortium (COSMIC)

COSMIC is a voluntary, not-for-profit organization of software metrics experts from around the world, founded in 1998. All its publications are 'open' and available for free distribution subject to copyright and acknowledgement restrictions. For more on COSMIC and its organization, see the COSMIC website www.cosmic-sizing.org .

The Measurement Practices Committee continues to work on improving the document so that it is easier to understand. We strongly urge readers and translators to contact the MPC to report any possible defects and/or text that is unclear, using the process of Appendix G.

The COSMIC Measurement Practices Committee

Table of Contents

INTRODUCTION

1.0   Chapter summary

This chapter has four purposes:

- To explain the types of software ('applicable domains') for which the COSMIC method may be used, and the limitations on its use.

- To define 'Functional User Requirements' ('FUR'), i.e. the requirements for software functionality that the COSMIC method aims to measure. We explain in general terms how a Measurer can extract or derive the FUR from available software artefacts for a functional size measurement. Non-Functional Requirements ('NFR') are also defined, since requirements that are initially expressed as non-functional often evolve as a project progresses partly or wholly into FUR that may also be measured.

- To define the basic principles of the COSMIC method, as summarized in two models. The 'Software Context Model' is used to characterize a piece of software to be measured. The 'Generic Software Model' defines the key principles of the COSMIC model of the FUR whose functional size is to be measured.

- To define the COSMIC method measurement process and measurement principle (related to the method's unit of measurement).

1.1   Applicability of the COSMIC method

The COSMIC method is designed to be applicable to measure the functionality of software from the following domains:

- Business application software that is typically needed in support of business administration, such as banking, insurance, accounting, personnel, purchasing, distribution or manufacturing, etc. Such software is often characterized as 'data rich', as it is dominated largely by the need to manage large amounts of data about events and 'things' in the real world.

- Real-time software, the task of which is to keep up with or control events happening in the real world. Examples would be software for telephone exchanges and message switching, software embedded in devices to control machines such as domestic appliances, elevators, vehicles and aircraft, for process control and automatic data acquisition, any software used in the 'Internet of Things', and software within the operating system of computers.

- Infrastructure software in support of the above such as re-usable components, device drivers and such-like.

- Some types of scientific/engineering software.

1.2   Functional User Requirements

The COSMIC method involves applying a set of models, principles, rules and processes to measure the Functional User Requirements (or 'FUR') of a given piece of software. The

result is a numerical 'value of a quantity' (as defined by ISO) representing the functional size of the piece of software according to the COSMIC method.

Functional User Requirements are defined by ISO [17] as follows.

DEFINITION – Functional User Requirements (FUR)

A sub-set of the user requirements. Requirements that describe what the software shall do, in terms of tasks and services.

NOTE 1: Functional User Requirements relate to but are not limited to:

- data transfer (for example Input customer data; Send control signal)

- data transformation (for example Calculate bank interest; Derive average temperature)

- data storage (for example Store customer order; Record ambient temperature over time)

- data retrieval (for example List current employees; Retrieve latest aircraft position)

Examples of user requirements that are not Functional User Requirements include but are not limited to:

- quality constraints (for example usability, reliability, efficiency and portability)

- organizational constraints (for example locations for operation, target hardware and compliance to standards)

- environmental constraints (for example interoperability, security, privacy and safety)

- implementation constraints (for example development language, delivery schedule)

NOTE 2: In COSMIC documents, the term 'FUR' is restricted to mean the functional user requirements that:

- are derived from the available software artefacts (requirements, designs, physical artefacts, etc.);

- are adjusted, if necessary, by assumptions to overcome uncertainties in the available artefacts;

- contain all the information needed for a COSMIC Functional Size Measurement.

Otherwise we use expressions such as 'functional requirements', 'actual requirements' or 'physical artefacts', etc., depending on the context.

Note that the COSMIC method recognizes that some types of requirements (e.g. quality and environmental constraints) may be expressed early in the life of a software project as 'Non-Functional' Requirements, consistent with this ISO definition. However, these same requirements may evolve as the project progresses into Functional User Requirements. See section 1.2.3 of this Measurement Manual.

Functional sizes measured by the COSMIC method are designed to depend only on the FUR of the software to be measured (including FUR that are derived from NFR) and to be independent of any requirements or constraints concerning the implementation of the FUR. 'Functionality' may be loosely defined as 'the information processing that the software must perform for its users'.

1.2.1 Extracting the Functional User Requirements from software artefacts

In the real world of software development it is rare to find 'artefacts' for the software such as documents that describe the software in detail, or visible manifestations of the software such as display screens, in which the FUR are clearly distinguished from other types of requirements and are expressed in a form suitable for direct measurement without any need for interpretation. This means that usually the Measurer will have to extract the FUR from the available artefacts of the software, before mapping them to the concepts of the COSMIC 'models of software'.

As illustrated in Figure 1.1, FUR can be derived from software engineering artefacts that are produced before the software exists. Thus, the functional size of software can be measured prior to its implementation in a computer system.

| **Requirements definition artefacts** e.g. Use Cases, User Stories | **Data analysis / modelling artefacts** e.g. E/R diagrams, RDA, OO models | **Dynamic behaviour models** e.g. State Transition Diagrams |

Functional User Requirements ('FUR') in the artefacts of the software to be measured
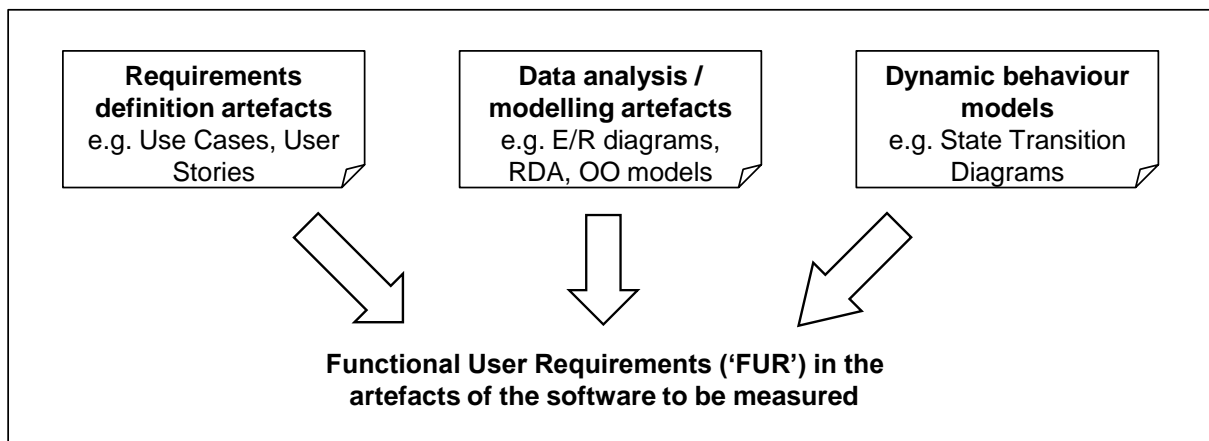
Figure 1.1 – Pre-implementation sources of Functional User Requirements

NOTE: Requirements may be produced even before they are allocated to hardware or software. Since the COSMIC method is aimed at sizing the FUR of a piece of software, only the FUR allocated to the software are measured. However, in principle COSMIC can be applied to requirements before they are allocated to software or to hardware, regardless of the eventual allocation decision. For example, it is straightforward to size the functionality of a pocket calculator using the COSMIC method without any knowledge of what hardware or software (if any) is involved. However, the assertion that the COSMIC method can be used to size requirements allocated to hardware needs more testing in practice before it can be considered as fully validated without the need for further rules.

In other circumstances, some existing software may need to be measured without there being any, or with only a few, architecture or design artefacts available, and the FUR might not be documented (e.g. for legacy software). In such circumstances, it is still possible to derive the FUR from the artefacts of the computer system even after it has been implemented, as illustrated in Figure 1.2.
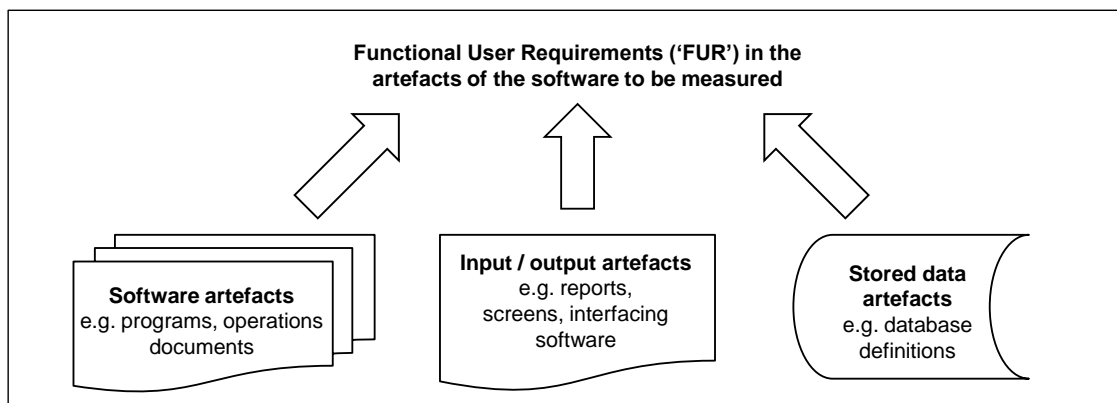
Figure 1.2 – Post-implementation sources of Functional User Requirements

1.2.2 The process of deriving Functional User Requirements from software artefacts

The process to be used and hence the effort required to extract the FUR from different types of software engineering artefacts or to derive them from installed software will obviously vary enormously; these processes cannot be dealt with in the Measurement Manual. The method assumes that the Functional User Requirements of the software to be measured either exist or that they can be extracted or derived from its artefacts, in light of the purpose of the measurement.

The Measurement Manual is therefore confined to defining and describing the concepts of the COSMIC models (the 'Software Context Model' and the 'Generic Software Model' – see section 1.3) and how to apply them to measure the FUR of a given piece of software.[2]

If the Measurer really understands these two models, it will always be possible to derive the FUR of a piece of software to be measured from its available artefacts, though the Measurer may well have to make some assumptions due to missing or unclear information.

1.2.3 Non-Functional Requirements

The ISO standard definition of Functional User Requirements (or 'FUR', see above) lists various types of 'user requirements' in Note 1 that are not FUR. By implication these are Non-Functional Requirements (NFR).

NFR can be very significant for a software project. In extreme cases, a statement of requirements for a software-intensive system can require as much documentation for the NFR as for the FUR. But the distinction between NFR and FUR is not as simple as it appears from the ISO definition of FUR. The COSMIC method can be used to measure some requirements that may be first expressed as non-functional. First we need to define NFR [18]:

---

[2] Various COSMIC Guidelines, e.g. for sizing business application software [7] and for sizing real-time software [4] give guidance on the mapping from various data analysis and requirements determination methods and from software artefacts to the concepts of COSMIC.

DEFINITION – Non-Functional Requirements (of software)
Any requirement for the software part of a hardware/software system or software product, including how it should be developed and maintained, and how it should perform in operation, except a functional user requirement for software. Non-functional requirements concern: • the software quality; • the environment in which the software must be implemented and which it must serve; • the processes and technology to be used to develop and maintain the software; • the technology to be used for the software execution. NOTE: System or software requirements that are initially expressed as non-functional often evolve as a project progresses wholly or partly into FUR for software.

Several studies [3] have shown that some requirements that initially appear as *system* NFR evolve as a project progresses into a mixture of requirements that can be implemented in software functions, and other requirements or constraints that are truly 'non-functional'. See Figure 1.3. This is true for many system quality constraints, such as response time, ease of use, maintainability, etc. Once identified, these software functions that have been 'hidden' in NFR at the beginning of a project can be sized using the COSMIC method just as any other software functions. Not recognizing this 'hidden' functional size is one reason why software sizes can appear to grow as a project progresses.
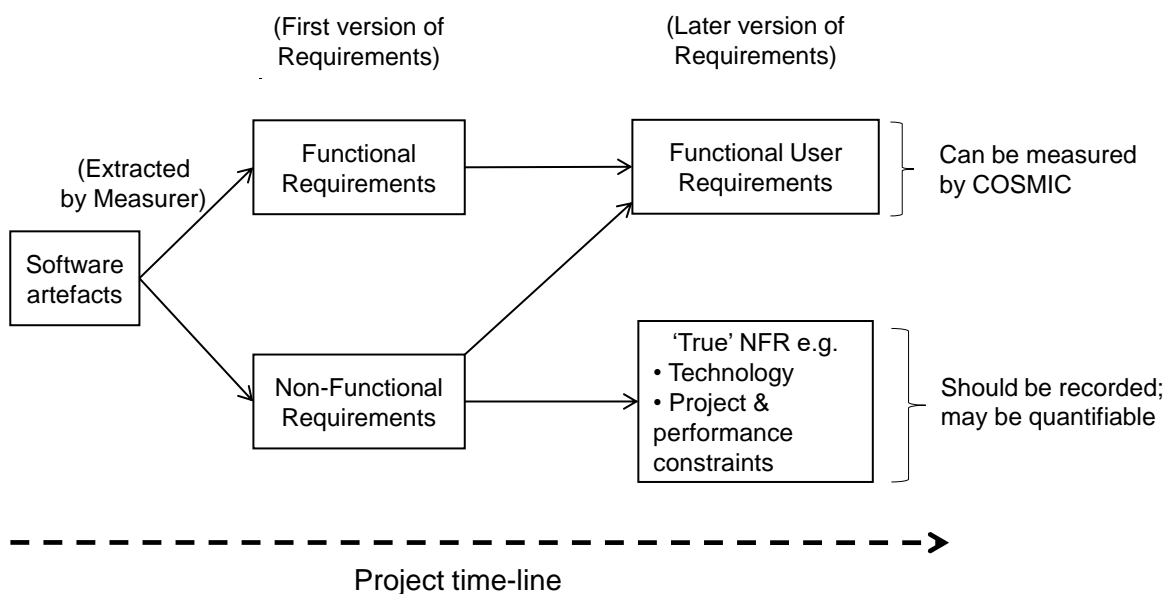
Figure 1.3 - Many requirements initially appearing as NFR evolve into FUR as a project progresses

BUSINESS EXAMPLE: The requirements for a new software system include the statement 'the user shall have the option to secure files by encryption'. The project to develop the system is at the stage of estimating effort and cost. Two options are considered:

• *Develop some proprietary encryption software. For project estimating purposes it may be necessary to measure the size of the FUR for the encryption software.*

- *Purchase an existing Commercial Off-The-Shelf (COTS) package. For project estimating purposes it may be necessary to measure only the size of the software functionality needed to integrate the COTS package. The cost of the package and the effort to integrate and test the file encryption package will also need to be considered in the project cost estimate.*

REAL-TIME EXAMPLE: Dependability or fault tolerance requirements for aerospace systems are achieved mostly through a combination of redundancy and backup of the physical systems. A function such as engine monitoring is implemented on two or more separate embedded computers. This function has a strict timing constraint stated as a NFR: 'each separate computer must respond within a specific time. If any one of the computers repeatedly responds later than the required time, or its results disagree with the others, it must be out-voted' (by a mechanism specified as a functional requirement). A requirement for fault tolerance that when initially stated may appear as non-functional therefore evolves into FUR that can be measured. The timing mechanism can also be partly implemented in software and this functionality can also be measured (see for example the 'Guideline for sizing real-time software' [4], section 3.2).

For more examples, see Appendix B. All these examples demonstrate that when there is a requirement to measure a size of some software early in the life of a project, it is important to consider whether some NFR could evolve into software FUR and whether the size of these software FUR should also be measured.

1.3   The fundamental principles of the COSMIC method

The COSMIC method is based on fundamental software engineering principles. These principles are summarized in two models.

In the same way that a house can have many sizes depending on what you want to measure, the size of a piece of software can be measured in many ways, even using the same unit of measurement. The principles of the 'Software Context Model' enable a Measurer to define the software to be measured and the size measurement. These ensure that the results can be understood and interpreted consistently by future users.

The principles of the 'Generic Software Model' define how the FUR of the software to be measured are modeled so that they can be measured.

The main reason for including these two models at this stage in the Measurement Manual is to show how the COSMIC method is fundamentally very simple. We will also need to refer to the two models later in the Manual. However, a novice Measurer should not expect to be able to read these two models and then go away and measure accurately. To apply the models to a particular measurement situation, the Measurer will need the definitions of the various concepts and the additional principles, rules, explanations and examples given in this Manual.

N.B. Terms that are given in bold when first used in the following sections 1.3.1 and 1.3.2 are used with meanings that may be specific to the COSMIC method. For the formal definitions, see the glossary in Appendix F of this Measurement Manual. The references given with each principle are to the sections of this Manual where the subject is dealt with in detail.

1.3.1 The COSMIC Software Context Model

PRINCIPLES – The COSMIC Software Context Model
a) Software is bounded by hardware.
b) Software is typically structured into **layers** (2.2.2).
c) A layer may contain one or more separate '**peer'** pieces of software (2.2.2).
d) Any piece of software to be measured, shall be defined by its **measurement scope**, which shall be confined wholly within a single layer (2.2).
e) The scope of a piece of software to be measured shall depend on the **purpose** of the measurement (2.1).
f) The **functional users** of a piece of software to be measured shall be identified from its **Functional User Requirements** (FUR) as the senders and/or intended recipients of data to/from the software respectively (2.3).
g) The functional requirements of software may be expressed at different **levels of granularity** (2.4).
h) A precise COSMIC size measurement of a piece of software requires that its FUR are known at the levels of granularity at which its **functional processes** and sub-processes can be identified (2.4.3).
i) If the functional requirements of a piece of software are available only at a high level of granularity, an approximation approach can be used to measure a size at the high level of granularity and to scale the result to an approximate COSMIC size at the levels of the functional processes and data movements. (2.4.3).

1.3.2 The Generic Software Model

Having identified and defined the FUR of the software to be measured in terms of the Software Context Model, we now apply the Generic Software Model to the FUR to identify the components of the functionality that will be measured. This Generic Software Model assumes that the following general principles hold true for any software that can be measured with the method.

PRINCIPLES – The COSMIC Generic Software Model
a) A piece of software interacts with its functional users across a **boundary**, and with **persistent storage** within this boundary (2.3).
b) Functional user requirements of a piece of software to be measured can be mapped into unique functional processes (3.2).
c) Each functional process consists of sub-processes (3.2).
d) A sub-process may be either a **data movement** or a data manipulation (3.2).
e) A data movement moves a single **data group** (3.3).
f) There are four data movement types, **Entry**, **Exit**, **Write** and **Read** (3.5).
• An Entry moves a data group into a functional process from a functional user.
• An Exit moves a data group out of a functional process to a functional user.
• A Write moves a data group from a functional process to persistent storage.

> - A Read moves a data group from persistent storage to a functional process.
>
> g) A data group consists of a unique set of **data attributes** that describe a single object of interest (3.3).
>
> h) Each functional process is started by its **triggering Entry** data movement. The data group moved by the triggering Entry is generated by a functional user in response to a **triggering event** (3.2).
>
> i) The size of a functional process is equal to the total count of its data movements.
>
> j) A functional process shall include at least the triggering Entry data movement and either a Write or an Exit data movement, i.e. it shall include a minimum of two data movements. There is no upper limit to the number of data movements in a functional process and hence no upper limit to its size (3.3)
>
> k) As an approximation for measurement purposes, data manipulation sub-processes are not separately measured; the functionality of any data manipulation is assumed to be accounted for by the data movement with which it is associated (3.5.6).
>
> NOTE: The COSMIC Generic Software Model, as its name suggests, is a logical 'model' that exposes units in which software processes data that are suitable for functional size measurement. The model does not intend to describe the physical sequence of the steps by which software executes nor any technical implementation of the software.

1.3.3 Types versus occurrences

To understand this Measurement Manual it is essential that the reader can distinguish between 'types' and 'occurrences'.

- In general, the 'type' of a thing is an abstract class of all the things that share some common characteristic. (Synonyms of 'type' are 'category' or 'kind'). In particular, in Functional Size Measurement, 'things' are of the same type if they share the same FUR.

- An 'occurrence' of a thing is when the thing appears in practice, e.g. in a real-world context, or when an event occurs, or when a process is executed by a person or a computer. (A synonym of 'occurrence' is 'instance'). An 'occurrence' is created when the characteristics of a 'type' of something are given actual values, known in the Object-Oriented world as 'instantiation' – the creation of an instance.)

All Functional Size Measurement methods define the 'types of things' that a Measurer must identify in any given functional requirements in order to measure a functional size.

In the case of the COSMIC method, these 'types of things' include functional user types and functional process types from the Software Context Model and all the concepts shown in bold, e.g. data movement types, object of interest types, etc., in the Generic Software Model.

For ease of reading, however, we normally omit 'type' when using these terms.

Examples from the Software Context Model

BUSINESS EXAMPLE 1: The system supporting a Call Centre has 100 employees who answer customer questions. A context model of the system would show one functional user type: 'Call Centre employee' of which there are 100 occurrences.

REAL-TIME EXAMPLE 2: The embedded software of a digital radio sends its output to a pair of stereo loudspeakers. The software sends separate signals (of the same type) to each of the two loudspeakers. They each convert the received electrical signal into sound

in the same way. A context model of the software would show one functional user type 'loudspeaker' of which there are two occurrences.

Examples from the Generic Software Model

BUSINESS EXAMPLE 3: Suppose a functional process (-type) that enables data to be entered and validated for a new customer. The Entry (-type) data movement will be executed, i.e. it will <u>occur once</u>, each time a human functional user registers data for a new customer. During its execution, the functional process must validate the entered data by searching to check if the customer already exists in the database. Hence the Read (-type) data movement for this validation will occur <u>one or more</u> times (depending on the database design). However, one Entry (-type) and one Read (-type) of the customer are counted when measuring this process.

REAL-TIME EXAMPLE 4: Suppose a functional process (-type) that must control the temperature of an oven once every ten seconds. The functional process will be executed, i.e. it will <u>occur,</u> once every 10 seconds. During its execution, the Exit (-type) data movement of the process to switch the heater on or off may or may not be executed i.e. it <u>may or may not occur</u> at all in any cycle, depending on whether the heater must be switched on or off, or left in its current state. The Exit (-type) data movement is counted once in the functional process, regardless of whether or not it occurs in a particular execution.

Note: The <u>number</u> of occurrences of any functional user, or of any Entry (-type), Exit (-type), Read (-type) or Write (-type) data movement when software is executed is irrelevant to the measurement of a COSMIC functional size. However, determining the <u>relative</u> frequency of data movement occurrences can help distinguish different data movement types in certain cases. (See for example the rule in section 3.3.2).

1.4   The COSMIC measurement process and the unit of measurement

The COSMIC measurement process consists of three phases:

- the Measurement Strategy phase, in which the purpose and scope of the measurement are defined. The Software Context Model is then applied so that the software to be measured and the required measurement are unambiguously defined. (Chapter 2)

- the Mapping Phase in which the Generic Software Model is applied to the FUR of the software to be measured to produce the COSMIC model of the software that can be measured. (Chapter 3)

- the Measurement Phase, in which actual sizes are measured. (Chapter 4)

Rules for how measurements should be recorded are given in Chapter 5.

The relationship of the three phases of the COSMIC method is shown in Figure 1.4:
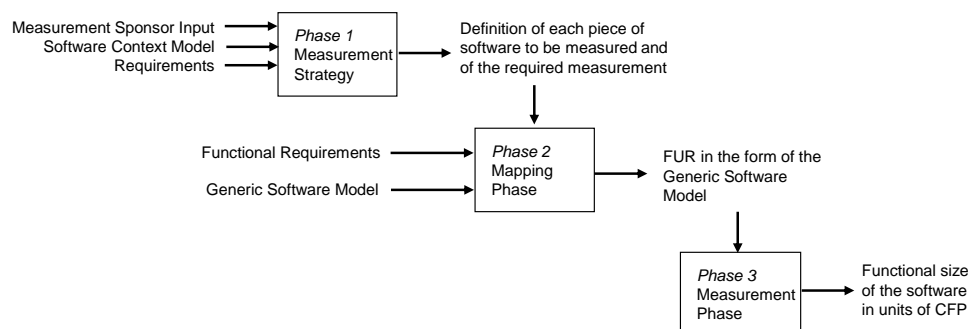
Figure 1.4 – The COSMIC method measurement process

The COSMIC unit of measurement (the 'CFP') and the measurement principle are defined as follows.

DEFINITION – COSMIC unit of measurement
1 CFP (Cosmic Function Point), which is the size of one data movement.

PRINCIPLE – The COSMIC measurement principle
a) The size of a functional process is equal to the number of its data movements.
b) The functional size of a piece of software of defined scope is equal to the sum of the sizes of its functional processes.

Sizes of required changes to a piece of software are measured as follows:

- The size of any data movement impacted (i.e. that must be added, modified or deleted) by the required change is measured by convention as 1 CFP.
- The size of the required changes to a piece of software is equal to the number of data movements that are impacted by the required changes.
- The minimum size of a change to a piece of software is 1 CFP.

Further rules and guidance on measurement and aggregation of measurements are given in sections 4.1 to 4.4 of this Measurement Manual.

1.5   Limitations on the applicability of the COSMIC method

See section 4.5 for possible limitations of the method and how it may be possible to extend the method locally to overcome the limitations.

THE MEASUREMENT STRATEGY PHASE

2.0 Chapter summary

This chapter describes the key parameters that must be considered in the first 'Measurement Strategy' phase of the measurement process, before actually starting to measure. These are (*in italics*):

- The *purpose* of the measurement, i.e. what the result will be used for. The purpose determines the other parameters of a measurement.

- The *overall scope* of the software to be measured and, if the software consists of more than one part that should be measured separately (e.g. the components of a distributed software system), the *measurement scope(s)* of the individual parts.

- The *layer(s)* of the software architecture in which each piece of the software to be measured resides.

- The *level of decomposition* of the piece(s) of software to be measured, e.g. a whole application or the level of its main components, or the level of re-used components from a Service-Oriented Architecture, etc.

- The *functional users* of each piece of software to be measured. These are the senders and intended recipients of data to/from the software to be measured; they may be humans, hardware devices or other pieces of software.

- The *level of granularity* of the available artefacts of the software to be measured. For example, are all requirements defined at the same level of detail? Do we have enough detail for a precise COSMIC measurement or only enough for an approximate measurement?

Determining these parameters helps answer the questions of 'which size should be measured', 'how accurate do we want the measurement', etc. Recording the parameters enables future users of a measurement to decide how to interpret the measurement.

It is important to note that these parameters and related concepts are not specific to the COSMIC FSM method, but should be common to all FSM methods. Other FSM methods may not distinguish different types of functional users and may not discuss different levels of granularity, etc. It is only the broader applicability and flexibility of the COSMIC method that requires these parameters to be considered more carefully than with other FSM methods.

It is very important to record the data arising from this Measurement Strategy phase (as listed in section 5.2) when recording the result of any measurement. Failure to define and to record these parameters consistently will lead to measurements that cannot be interpreted reliably and compared, or be used reliably as input for processes such as estimating project effort.

The sections of this chapter give the formal definitions, principles and rules and some examples for each of the key parameters to help the Measurer through the process of determining a measurement strategy, as shown in Figure 2.0 below.

Each section gives some background explanation of why the key parameter is important, using analogies to show why the parameter is taken for granted in other fields of

measurement, and hence must also be considered in the field of software functional size measurement.

Note from Figure 2.0 that determining the parameters of the Measurement Strategy may need some iteration.
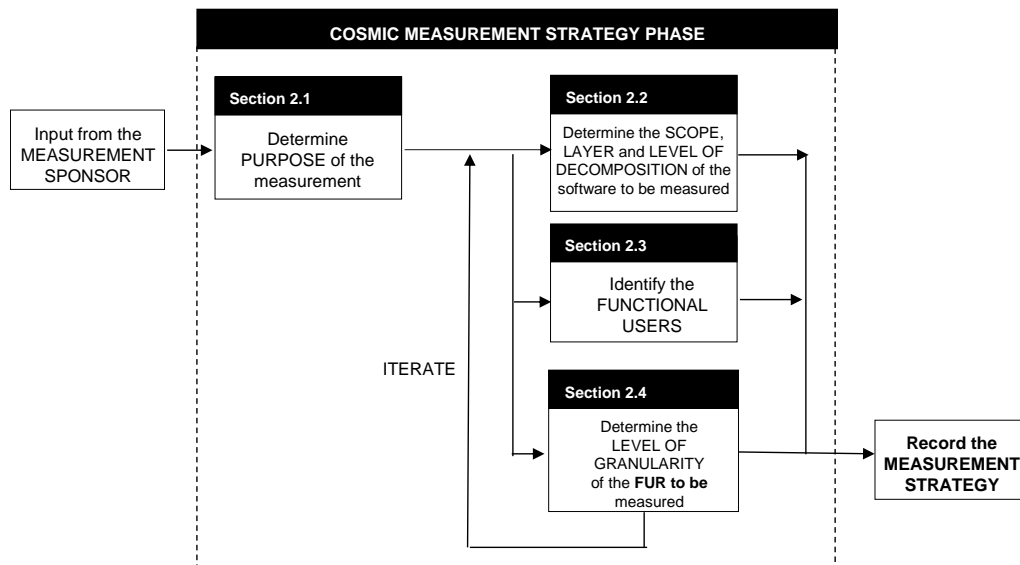
Figure 2.0 - The process of determining a Measurement Strategy

Measurement Strategy Patterns

As an aid to determining a measurement strategy, the Guideline for 'Measurement Strategy Patterns' [5] describes, for each of several different types of software, a standard set of parameters for measuring software sizes, called a 'measurement strategy pattern' (abbreviated to 'measurement pattern').

DEFINITION – Measurement (Strategy) Pattern
A standard template that may be applied when measuring a piece of software from a given software functional domain, that defines the types of functional user that may interact with the software, the level of decomposition of the software and the types of data movements that the software may handle.

Consistent use of the same measurement patterns should help Measurers to ensure that measurements made for the same purpose are made in a consistent way, may be safely compared with other measurements made using the same pattern and will be correctly interpreted for all future uses. A side benefit of using a standard pattern is that the effort to determine the Measurement Strategy parameters is much reduced. We do, however, strongly recommend that Measurers study and master the COSMIC method, especially the Measurement Strategy parameters, before using the standard patterns.

2.1 Defining the purpose of the measurement

The term 'purpose' is used in its normal English meaning.

DEFINITION – Purpose of a measurement
A statement that defines why a measurement is required, and what the result will be used for.

2.1.1 The purpose of the measurement – an analogy

There are many reasons to measure the functional size of software, just as there are many reasons to measure, say, the surface areas of a house and in both cases the different reasons may result in different sizes. Starting with the analogy, the size of a house may well vary with:

- the reason for and timing of the measurement (e.g. depending on the need to measure the customer's outline specification for budgeting, or the architect's plans for a precise cost estimate, or the actual size on completion for planning floor coverings),
- the artefacts measured (e.g. the plans or the physical building).

Note, however, the same measurement principles and the unit of measurement are used for all the measurements. In exactly the same way, the measured size of a piece of software may well vary with:

- the reason for and timing of the measurement (e.g. depending on the need to measure prior to development for estimating purposes, or during development to track scope creep, or after installation to measure the developer's performance),
- the artefacts measured (e.g. a statement of requirements, or the physical software artefacts).

As with the analogy, however, the same measurement principles and the unit of measurement are used for all the measurements.

Clearly the Measurer of a piece of software must decide, depending on the purpose of the measurement, *when* to measure (before, during or after development), *what* to measure (e.g. all the software to be delivered by a project, or to exclude re-used software) and *which artefacts* to use to derive the FUR to be measured (e.g. a statement of requirements or the installed software).

> *EXAMPLES: The following are typical measurement purposes*
>
> - *To measure the size of the FUR as they evolve, as input to a process to estimate development effort.*
>
> - *To measure the size of changes to the FUR after they have been initially agreed, in order to manage project 'scope creep'.*
>
> - *To measure the size of the FUR of the delivered software as input to the measurement of the development organization's performance.*
>
> - *To measure the size of the FUR of the total software delivered, and also the size of the FUR of the software which was developed, in order to obtain a measure of functional re-use.*
>
> - *To measure the size of the FUR of the existing software as input to the measurement of the performance of the group responsible for maintaining and supporting the software.*
>
> - *To measure the size of some changes to (the FUR of) an existing software system as a measure of the size of the work-output of an enhancement project team.*
>
> - *To measure the size of the sub-set of the total software functionality that must be developed, that will be provided to the software's human functional users.*

2.1.2 The importance of the purpose

The purpose helps the Measurer to determine:

- The scope to be measured and hence the artefacts which will be needed for the measurement.

- The functional users (as will be shown in section 2.3, the functional size changes depending on who or what is defined as the functional user).

- The point in time in the project life-cycle when the measurement will take place.

- The required accuracy of the measurement, and hence whether the standard COSMIC method should be used, or whether an approximation version of the standard COSMIC method should be used (e.g. early in a project's life-cycle, before the FUR are fully elaborated).

Both of these latter two points will determine the level of granularity at which the FUR will be measured.

2.2 Defining the scope of the measurement

DEFINITION – Scope of a measurement
The set of Functional User Requirements to be included in a specific functional size measurement exercise.
NOTE: (specific to the COSMIC method) A distinction should be made between the 'overall scope', i.e. all the software that should be measured according to the purpose, and the 'scope' of any individual piece of software within the overall scope, whose size should be measured separately. In this Measurement Manual, the term 'scope' (or the expression 'measurement scope') will relate to an individual piece of software whose size must be measured separately.

RULES –Scope of a measurement
a) The scope of any piece of software to be measured shall be derived from the purpose of the measurement.
b) The scope of any one measurement shall not extend over more than one layer of the software to be measured.

See the next section for examples of the overall scope and measurement scopes.

2.2.1 Deriving the measurement scope from the measurement purpose

The software defined by an *overall scope* may be sub-divided into individual pieces of software each with their own defined *measurement scope* in many ways, depending on the purpose of the measurement. Suppose an overall scope is defined as 'the application portfolio of organization X' or as 'all the pieces of software to be delivered by project Y'. The sub-divisions could be made due to:

- software in different layers (due to rule b) above),

- different organizational responsibilities, e.g. by customer-group or sub-project team,

- the need to distinguish different deliverables for performance measurement, effort estimation or for software contract purposes.

The latter reason could be due to the need to distinguish separate measurement scopes for pieces of software that:

- are built using different technologies, i.e. hardware platform, programming language, etc.,

- operate in different modes, i.e. on-line-versus batch modes,

- are developed as opposed to 'delivered' (the latter including package-implemented or other re-used software),

- are at different levels of decomposition, e.g. a whole application or a major component or a minor component such as re-usable object,

- are the main deliverables as opposed to software that is used once, e.g. for data conversion, and then discarded; it might not be worth the effort of measuring the latter,

- are delivered by single 'sprints' of an agile process

- are developed versus enhanced software,

and any combinations of these factors.

In summary, the purpose of the measurement must always be used to determine (a) what software is included or excluded from the overall scope and (b) the way the included software may need to be divided up into separate pieces, each with its own scope, to be measured separately.

In practice a scope statement needs to be explicit rather than generic, e.g. the developed work-output of project team 'A', or application 'B', or the portfolio of enterprise 'C'. The scope statement may also, for clarity, need to state what is excluded.

BUSINESS EXAMPLE: Figure 2.1 shows all the separate pieces of software – the 'overall scope' - delivered by a project team:

- *the client and the server components of an implemented application software package*

- *a program that provides an interface between the server component of the new package and existing applications*

- *a program that is used once to convert existing data to the new format required by the package. This program was built using a number of re-usable objects developed by the project team*

- *the device driver software for new hardware on which the package client component will execute*

(Each individual piece of software for which a measurement scope was defined is shown as a solid rectangular box.)
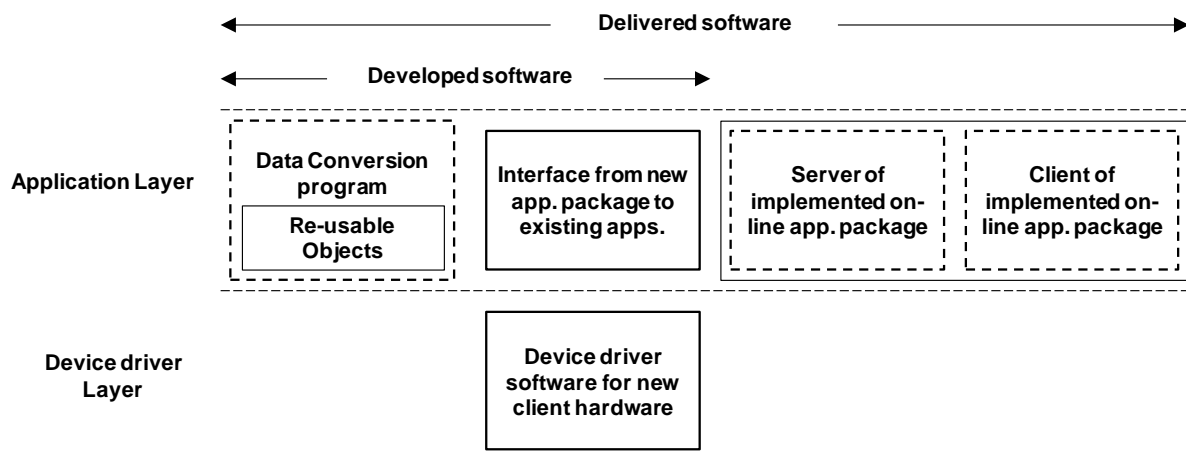
Figure 2.1 - The overall scope of the deliverables of a software project and the individual measurement scopes

The diagram shows that the 'delivered' pieces of software consisted of some that were newly developed and some that were implemented by the project team. The purpose is to measure the FUR of the individual pieces of delivered software to be added to the size of the organization's software portfolio, considering the software package as a whole, i.e. ignoring the client-server component structure.

The size of the implemented package was added with that of the interface program to update the total size of the organization's application portfolio. The size of the data conversion program was not of interest as it was used once and thrown away. But the size of each of the re-usable objects was recorded in the organization's infrastructure software inventory, as well as that of the new device driver. Again these were classified separately.

Due to the diverse nature of the deliverables it would not be sensible when measuring the overall project team's performance to add together the sizes of all the delivered software. The performance of the teams that delivered each piece of software should be measured separately.

Note also that it is normally only of interest to measure the FUR of the software resulting from implementing a package, not the FUR of the package itself. The latter is perhaps only of interest to the package supplier.

2.2.2 Layers

Since the scope of a piece of software to be measured must be confined to a single software layer, the process of defining the measurement scope(s) may require that the Measurer first has to decide what are the layers of the software's architecture. In this section we will therefore define and discuss 'layers' of software as these terms are used in the COSMIC method. The reasons why we need these definitions and rules are as follows.

- The Measurer may be faced with measuring some software in a 'legacy' environment of software that evolved over many years without ever having been designed according to an underlying architecture (the software to be measured has a so-called 'spaghetti architecture'). The Measurer may therefore need guidance on how to distinguish layers according to the COSMIC terminology.

- The expressions 'layer' and 'layered architecture' are not used consistently in the software industry. If the Measurer must measure some software that is described as being in a 'layered architecture', it is advisable to check that 'layers' in this architecture are defined in a way that is compatible with the COSMIC method. To do this, the Measurer should establish the equivalence between specific architectural objects in the 'layered architecture' paradigm and the concept of layers as defined in this manual.

Layers may be identified according to the following definitions and principles.

DEFINITION – Layer
A functional partition of a software system architecture.

In a defined software architecture, each layer should comply with the following principles:

PRINCIPLES – Layer
a) Software in one layer provides a set of services that is cohesive according to some defined criterion, and that software in other layers can utilize without knowing how those services are implemented.
b) The relationship between software in any two layers is defined by a 'correspondence rule' which may be either • 'hierarchical', i.e. software in layer A is allowed to use the services provided by software in layer B but not vice versa (where the hierarchical relationship may be up or down), or • 'bi-directional', i.e. software in layer A is allowed to use software in layer B, and vice versa.
c) Software in one layer exchanges data groups with software in another layer

> via their respective functional processes.
>
> d) Software in one layer does not necessarily use all the functional services supplied by software in another layer.
>
> e) Software in one layer of a defined software architecture may be partitioned into other layers according to a different defined software architecture.

A measurement may relate to two or more 'peer' pieces of software, defined as follows:

DEFINITION – Peer pieces of software
Two pieces of software are peers of each other if they reside in the same layer.

EXAMPLE: The pieces of software in the application layer of Figure 2.1 are all peers of each other.

If the overall scope of the software to be measured extends over multiple layers, the Measurer should proceed as follows.

- If the software to be measured exists within an established architecture of layers that can be mapped to the COSMIC layering principles as defined above, then that architecture should be used to identify the layers for measurement purposes.

- If however, the purpose requires that some software is measured that is not structured according to the COSMIC layering principles, the Measurer should try to partition the software into layers by applying the principles defined above. Conventionally, infrastructure software packages such as database management systems, operating systems or device drivers, that provide services that can be used by other software in other layers, are each located in separate layers.

Normally in software architectures, the 'top' layer, i.e. the layer that is not a subordinate to any other layer in a hierarchy of layers, is referred to as the 'application' layer. Software in this application layer relies on the services of software in all the other layers for it to perform properly. Software in this 'top' layer may itself be layered, e.g. as in a 'three-layer architecture' of User Interface, Business Rules and Data Services components (see Business Example 5 below).

Once identified, each layer can be registered in the Generic Software Model matrix (Appendix A), with the corresponding label.

BUSINESS EXAMPLE 1: The physical structure of a typical layered software architecture (using the term 'layer' as defined here) supporting business applications software is given in Figure 2.2:
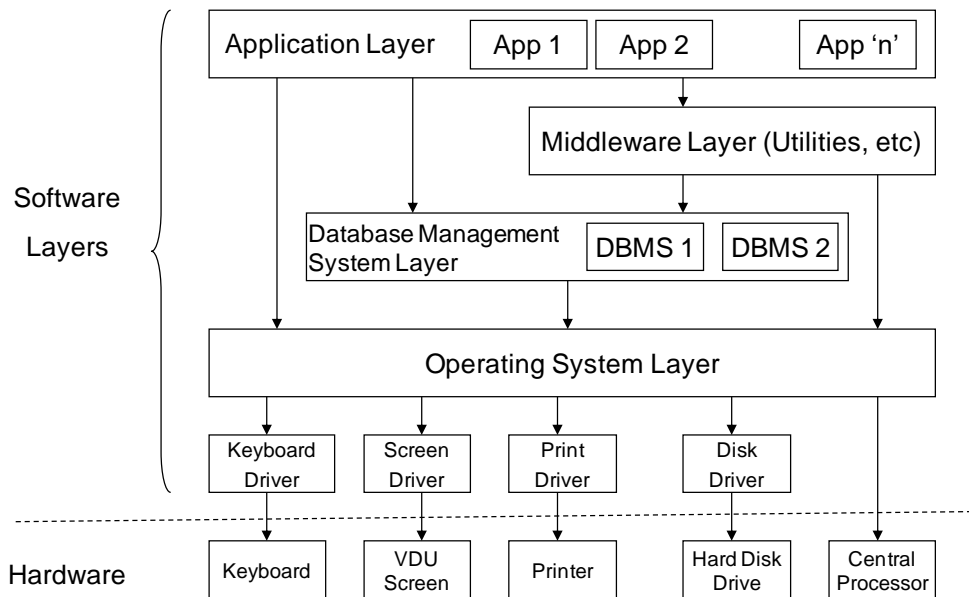
Figure 2.2- Typical layered software architecture for a Business/MIS system

REAL-TIME EXAMPLE 2: The physical structure of a typical layered software architecture (again using the term 'layer' as defined here) supporting a piece of embedded real-time software is given in Figure 2.3.(Note: simple uni-tasked real-time embedded software may not need a real-time operating system.)
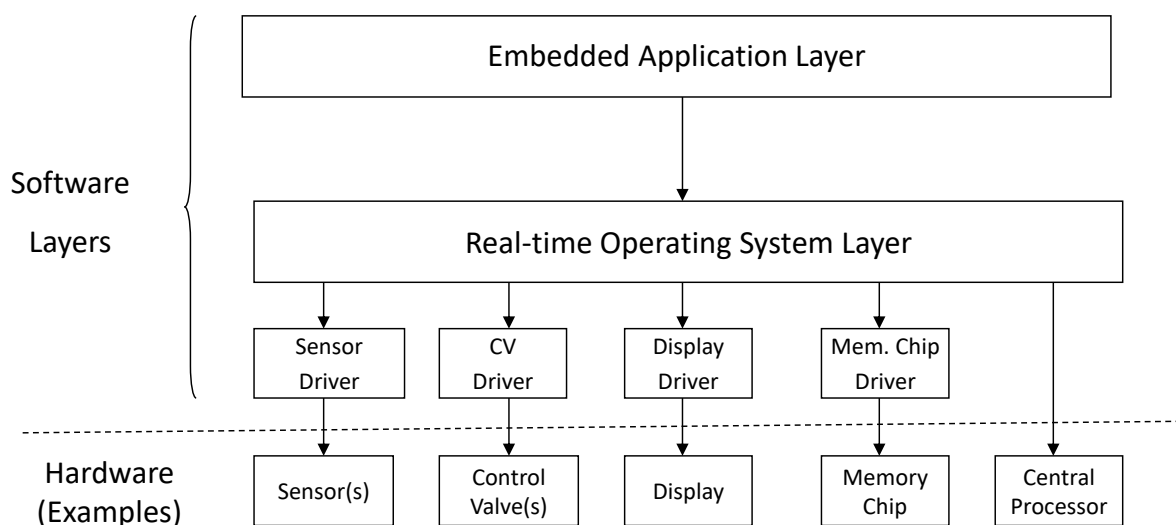
Figure 2.3 - Typical layered architecture for a real-time embedded-software system

REAL-TIME EXAMPLE 3: The ISO 7-layer (OSI) model for telecommunications. This defines a layered architecture for which the hierarchical correspondence rules for the layers of the message-receiving software are the inverse of the rules for the layers of the message-transmitting software.

REAL-TIME EXAMPLE 4: The 'AUTOSAR' architecture of the auto industry [19] which exhibits all the different types of correspondence rules between layers now described in the principles for a layer.

A software architecture may exhibit different layers depending on the 'view' of the architecture.

BUSINESS EXAMPLE 5: Consider an application A situated in a layered software architecture, as in Figure 2.4 below, which shows three possible layer structures a), b) and c) according to different architecture 'views'

a) View of an application 'A' as a whole	b) Application 'A' components in a 3-layer Architecture	c) Layers for SOA components of Business Rules

Application Layer

Application 'A'	UI Layer → User Interface Component	
	BR Layer → Business Rules Component	Application Layer / Orchestration Layer / Utility Layer
	DS Layer → Data Services Component	
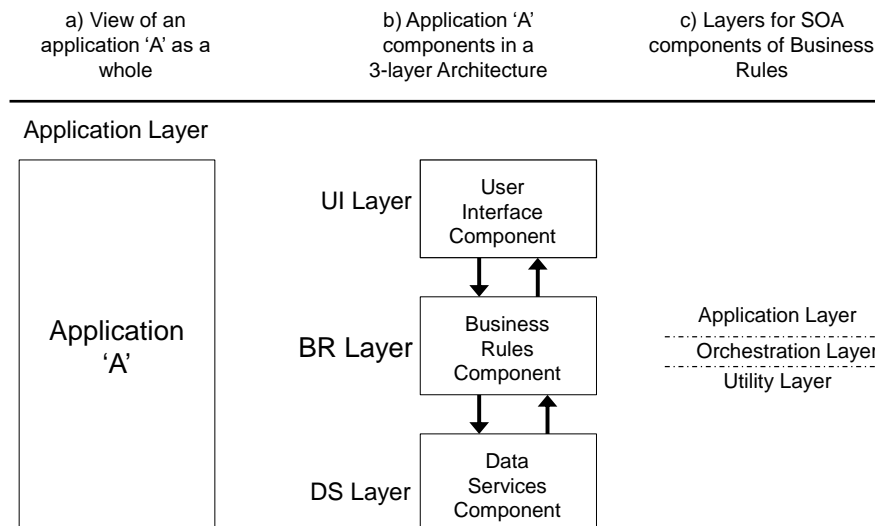
Figure 2.4 - Three views of the layers of an application

Purpose 1 is to measure the functional size of application A 'as a whole', as in View a). The measurement scope is the whole of application A, which exists entirely within the one 'application' layer

Purpose 2. Application A has been built according to a 'three-layer' architecture comprising a User Interface, Business Rules and Data Services components. Purpose 2 is to measure the three components separately as in View b). Each component is situated in its own layer of the architecture and the measurement scope must be defined separately for each component.

Purpose 3. The Business Rules component of the application has been built using re-usable components of a Service-Oriented Architecture, which has its own layer structure. Purpose 3 is to measure an SOA component of the Business Rules component as in View c). Each SOA component is situated in a layer of the SOA architecture and the measurement scope must be defined separately for each SOA component. (Note that SOA terminology also uses 'application layer' within its own architecture.)

2.2.3   Levels of decomposition

The 'level of decomposition' of a piece of software is defined as follows:

DEFINITION – Level of decomposition
Any level resulting from dividing a piece of software into components (named 'Level 1', for example), then from dividing components into sub-components ('Level 2'), then from dividing sub-components into sub-sub components ('Level 3'), etc.
NOTE 1: Not to be confused with 'level of granularity' which concerns the level of detail of requirements.
NOTE 2: Size measurements of the components of a piece of software are only

> directly comparable for components at the same level of decomposition.
>
> NOTE 3: Different levels of decomposition of a piece of software may correspond to different 'views' of the software's layers, e.g. as in Figure 2.4. However, software may be decomposed into 'levels' regardless of whether or not it is designed using a layered-architecture model.

Note 2 in the above definition is important because sizes of pieces of software at different levels of decomposition cannot be simply added up without taking into account the aggregation rules of section 4.3.1. Further, as a consequence, the performance (e.g. the productivity = size/effort) of projects to develop different pieces of software can only safely be compared if all the pieces of software are at the same level of decomposition.

EXAMPLE: The Guideline for 'Measurement Strategy Patterns' [5] recognizes three standard levels of decomposition: 'Whole application', 'Major Component' and 'Minor component'. See business example 5 in section 2.2.2 of this Measurement Manual, where the three levels are shown in Figure 2.4.

2.2.4 Defining the measurement scope: summary

Determining a measurement scope may involve more than just simply deciding what functionality should be included in the measurement. The decision may also involve consideration of the layer(s) in which the software to be measured resides and the level of decomposition of software at which the measurement(s) will be made, all dependent on the purpose of the measurement.

2.3 Identifying the functional users and recognizing persistent storage

A 'user' is defined, in effect[3], as 'any thing that interacts with the software being measured'. This definition is too broad for the needs of the COSMIC method. For the COSMIC method, the choice of user (or users) is determined by the Functional 'User' Requirements that must be measured and the purpose of the measurement. This (type of) user, known as the 'functional user', is defined as follows.

> **DEFINITION – Functional user**
>
> A (type of) user that is identified in the Functional User Requirements of a piece of software being measured as a sender and/or an intended recipient of data processed by that software.

. The choice of functional user types usually depends on the software domain.

- In the domain of business application software, the functional users are normally humans, plus perhaps other peer applications with which the application interfaces.

- For real-time software, the functional users would normally be engineered hardware devices that interact directly with the software, plus perhaps other peer software with which the real-time software interacts.

Note: it is normally strongly advised NOT to measure a size of the functionality of software as seen by a mixture of humans and hardware device functional users. The resulting size will be very difficult to interpret.

Note that the total set of 'users', i.e. including 'any thing that interacts with the software', must include the operating system. But the FUR of any application software would never include the operating system as a user. Any requirements that the operating system may impose on an application will be common to all applications, will normally be handled by the compiler or

---

[3] See the glossary for the definition, taken from ISO/IEC 14143/1:2007

interpreter, are invisible to the real functional users of the application and therefore do not appear in the FUR. In practical functional size measurement, an operating system never normally needs to be considered as a functional user of an application.

(However, if the purpose is to measure a software component of an operating system such as a device driver, then the operating system that calls the driver would be its functional user.)

RULES – Functional users
a) The functional users of a piece of software to be measured shall depend on the purpose of the measurement.
b) When the purpose of a measurement of a piece of software is related to the effort to develop or modify the piece of software, then the functional users should be all the different types of senders and/or intended recipients of data to/from the new or modified functionality, as required by its FUR.
NOTE: FUR may specify that multiple occurrences of functional users must be individually identified. Nevertheless they will be of the same type if each occurrence is subject to the same FUR.

BUSINESS EXAMPLE 1 illustrating rule b): In an order system a number of employees (human functional users) maintain the order data. An employee's ID is added to all data groups they enter. Identify one 'employee' functional user type because the order system FUR are the same for all these employees.

REAL-TIME EXAMPLE 2 illustrating rule b): Each wheel of a car has a sensor which obtains the pressure of its tire. At regular intervals, a functional process must obtain the pressure of all four tires. If the pressure is too low or too high - the range of safe pressures is in the software - the software shows which tire has a pressure problem on a diagram of the four wheels on a display screen at the dashboard. The functional users are the four sensors and the display screen. However, the four sensors are of the same type, so identify one functional user type 'sensor' and one functional user type for the display screen.

Note: The pressure-sensors of the four tires all work in the same way; the data group type they send is the same and the processing of this data is identical. Obviously, the software must be able to distinguish the four tires, (e.g. by using the sequence in which they report tire pressures when requested, or by their ID (e.g. 1 to 4) that they send with their pressure, etc.) in order to show on the display the pressure associated with each tire. But the FUR for the processing of data from each tire is the same, so there is only one functional user type: 'tire'.

2.3.1 Functional size may vary with the functional users

It is not always the case that the functional users are obvious. Different types of users of a 'thing' may 'see' different functionality and hence may measure different sizes of the 'thing'. In the case of software, different (types of) functional users may require (via their FUR) different functionality and therefore functional sizes will vary with the choice of functional users.

BUSINESS EXAMPLE 1: A software system has functionality to maintain basic personal data accessible to all staff of the Personnel Department, and more sensitive salary data accessible to only a sub-set of these staff. So this software has two types of functional users. The purpose of a measurement of this software should define whether the measurement scope applies to its total functionality accessible to all staff or is restricted to the functionality available to one of the two types of staff.

REAL-TIME EXAMPLE 2: Consider the embedded software of a copier. Excluding the operating system of the copier as a possible functional user, the software's functional users could be defined in one of two ways. They could be either (a) the human user who wants to make copies, or (b) the copier's hardware devices i.e. the control buttons, a screen on which messages are displayed to the human user, the paper transport mechanism, the paper jam sensors, the ink controller, indicator lights,, etc., with which the software interacts directly. These two types of functional users, humans or the set of hardware devices, will 'see' different functionality. The human user, for example, will be aware of only a sub-set of the total copier software functionality.

The developers of the embedded software that drives the copier will need to define the hardware devices as its functional users. Alternatively, a marketing person may find it useful to measure a size of the functionality of his own company's copier as seen by a human functional user versus that of a competitor's product in order to compare their price/performance[4].Do NOT try to mix the two views; a size measurement from a 'mixed' human/hardware view would be very difficult to interpret.

Having identified the functional users, it is then straightforward to identify the boundary. The boundary lies between the piece of software being measured and its functional users. We ignore any other hardware or software in that intervening space[5].

DEFINITION – Boundary
A conceptual interface between the software being measured and its functional users.
NOTE: It follows from the definition that there is a boundary between any two pieces of software in the same or different layers that exchange data where one piece of software is a functional user of the other, and/or vice versa.

NOTE: This definition of 'boundary' is taken from ISO/IEC 14143/1:2007, modified by the addition of 'functional' to qualify 'user'. To avoid ambiguity, note that the boundary should not be confused with any line that might be drawn around some software to be measured to define the measurement scope. The boundary is not used to define the scope of a measurement.

2.3.2   Persistent storage

DEFINITION – Persistent storage
Storage which enables a functional process to store a data group beyond the life of the functional process and/or from which a functional process can retrieve a data group stored by another functional process, or stored by an earlier occurrence of the same functional process, or stored by some other process.
NOTE 1: In the COSMIC model, persistent storage is a concept that exists only within the boundary of the software being measured, it cannot therefore be considered as a functional user of the software being measured.
NOTE 2: An example of 'some other process' would be the manufacture of read-only memory.

(For the definition of a 'functional process', see section 3.2).

---

[4] Toivonen, for example, compared the size of the functionality of mobile phones available only to human users in 'Defining measures for memory efficiency of the software in mobile terminals', International Workshop on Software Measurement, Magdeburg, Germany, October 2002.

[5] In fact, if the measurer has had to examine the FUR in order to identify the senders and intended recipients of data, the boundary will have already been identified.

Most often when analyzing the FUR for some software, if the Measurer finds requirements to store data in, or to retrieve data from a specific file or database, the FUR are not concerned with where or how the file/database is physically stored; the latter are technical requirements. All the FUR need to make clear is that the data is stored. In the Generic Software Model, the place where data is stored is 'persistent storage'. Persistent storage does not need to be identified; it is available to software in any layer.

An exception to this general guidance is if the FUR of the software being measured specify that the data be moved directly to or from an identifiable physical hardware storage device. ('Directly' means 'without passing through any intervening software'.) If the software being measured directly interacts with a hardware device, the device must be a functional user of that software; it should not be regarded as persistent storage. For more on this, see section 3.5.8.

2.3.3   Context diagrams

It can be very helpful when defining a measurement scope and the functional users to draw a 'context diagram' for the software being measured. In this and other COSMIC Guidelines, context diagrams are used to show the scope of a piece of software to be measured within its context of functional users (humans, hardware devices or other software) and the movements of data between them. (Context diagrams usually also show persistent storage, if relevant.)

A context diagram is effectively an instance of a measurement pattern (see section 2.0) applied to the software being measured. The key to symbols used in context diagrams is as indicated in Figure 2.5:
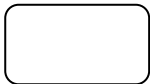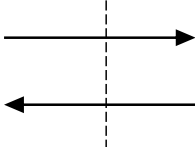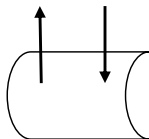
Symbol	Interpretation
(box with thick outline)	The piece of software to be measured (box with thick outline) i.e. the definition of a measurement scope.
(rounded box)	Any functional user of the software being measured.
(arrows crossing dotted line)	The arrows represent <u>all</u> the movements of data crossing a boundary (the dotted line) between a functional user and the software being measured.
(data storage symbol with arrows)	The arrows represent <u>all</u> the movements of data between the software being measured and 'persistent storage'. (The standard flowchart symbol for 'data storage' emphasizes that persistent storage is an abstract concept. Use of this symbol indicates that the software does not interact directly with physical hardware storage.)

Figure 2.5 - Key to the symbols of context diagrams

BUSINESS EXAMPLE: Figure 2.6 shows the context diagram for the client/server software of the implemented application package as in the Example shown in Figure 2.1 of section 2.2.1, to be measured as a 'whole', i.e. the fact that the application package has two components (client and server) is to be ignored for this measurement of the package.
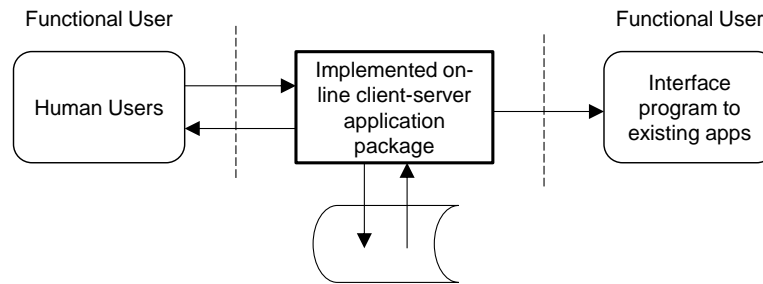
Figure 2.6 - Context diagram for the client-server application from section 2.2.1

REAL-TIME EXAMPLE: Figure 2.7 shows the context diagram for a simple intruder alarm embedded software system (taken from the 'COSMIC Guideline for sizing real-time software' [4], version 1.1, section 4.3)
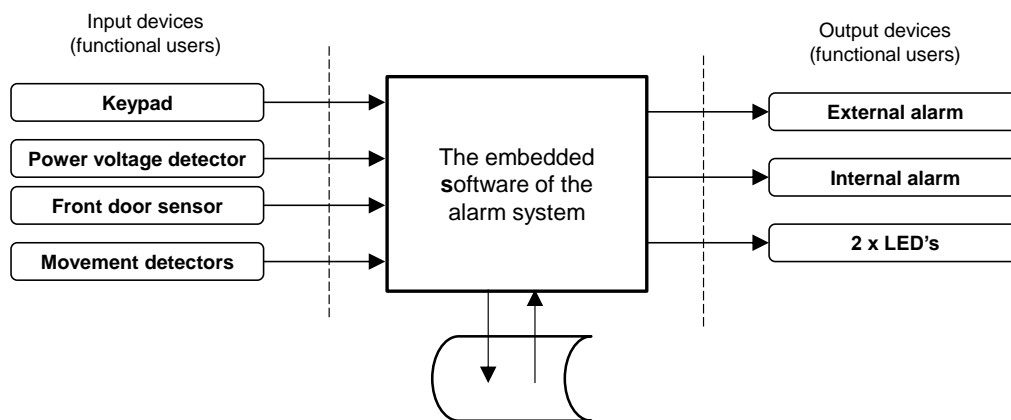
Figure 2.7 - Context diagram for the embedded software of an intruder alarm system

2.4   Identifying the level of granularity

2.4.1   The need for a standard level of granularity

In the initial stages of a software development project, actual requirements are specified 'at a high level', that is, in outline, or in little detail. As the project progresses, the actual requirements are refined, (e.g. through versions 1, 2, 3 etc.), revealing more and more detail 'at lower levels'. These different degrees of detail of the actual requirements are known as different 'levels of granularity'. (See also section 2.4.2 for other terms that may be confused with the concept of 'level of granularity' as defined here.)

DEFINITION – Level of granularity
Any level of expansion of the description of any part of a single piece of software (e.g. a statement of its requirements, or a description of the structure of the piece of software) such that at each increased level of expansion, the description of the functionality of the piece of software is at an increased and uniform level of detail.
NOTE: Measurers should be aware that when requirements are evolving early in the life of a software project, at any moment different parts of the required software functionality will typically have been documented at different levels of granularity.

In most product development activities, plans are drawn to standard scales, and it is easy to translate dimensions measured on one drawing to those on another drawing with a different scale. In contrast there are no standard scales for the various levels of granularity at which software may be specified, so it may be difficult to be sure that two statements of functional requirements are at the same level of granularity. Without agreement on some standard level of granularity at which to measure (or to which measurements should be scaled) it is impossible to know for sure that two functional size measurements may be compared.

To illustrate the problems further, consider this analogy. A set of road maps reveals the details of a national road network at three levels of granularity:

- map A shows only motorways and main highways;

- map B shows all motorways, main and secondary roads (as in an atlas for motorists);

- map C shows all roads with their names (as in a set of local district road maps).

If we did not recognize the phenomenon of different levels of granularity, it would appear that these three maps revealed different sizes of the nation's road network. Of course, with road maps, everyone understands the different levels of detail shown and there are standard scales to interpret the size of the network revealed at any level. The abstract concept of 'level of granularity' lies behind the scales of these different maps.

For software measurement, there is only one standard level of granularity that it is possible to define unambiguously. That is the level of granularity at which individual functional processes and their data movements can be identified and defined. Measurements should be made at this level or scaled to this level whenever possible[6].

2.4.2 Clarification of 'level of granularity'

Before proceeding further, it is important to ensure there is no misunderstanding about the meaning of 'level of granularity' in the COSMIC method. Zooming in on the functional requirements involves expanding the *description* of some software from a 'higher' to a 'lower' level of granularity and revealing more detail, *without changing its scope*. This process should NOT be confused with any of the following.

- Zooming in on some software in order to reveal its components, sub-components, etc. (at different 'levels of decomposition' – see section 2.2.3 above). Such zooming in may be required if the measurement purpose requires the overall measurement scope to be sub-divided following the physical structure of the software.

- Evolving the description of some software as it progresses through its development cycle, e.g. from requirements to logical design, to physical design, etc. Whatever the stage in the development of some software, we are only interested in its FUR for measurement purposes.

The concept of 'level of granularity' is therefore intended to be interpreted as applying only to the description of the requirements of software.

2.4.3 The standard functional process level of granularity

The 'functional process level of granularity' is the only level of granularity of functional requirements which it is possible to define as a standard. From this level, we can define a set of concepts that can result in repeatable, and therefore comparable size measurements[7].

---

[6] The topic of scaling measurements from one level of granularity to another is dealt with in the 'Guideline for early or rapid COSMIC functional size measurement by approximation approaches' [6] .

[7] The reason for the name 'functional process level of granularity' is that this is the level at which functional processes are identified – see section 3.2 for a more detailed discussion of functional processes.

DEFINITION - Functional process level of granularity

Any level of granularity in the description of a piece of software at which:

- its functional user (-types) are individual humans or engineered devices or pieces of software (and not any groups of these) AND

- single event (-types) occur that the piece of software must respond to (and not any level of granularity at which groups of events are defined).

NOTE 1: In practice, software documentation often describes functional requirements of different parts of the software at different levels of granularity, especially when the documentation is still evolving. Functional processes may be revealed at any of these different levels of granularity

NOTE 2: 'Groups of these' (functional users) might, for example, be a 'department' whose members handle many types of functional processes; or a 'control panel' that has many types of instruments; or 'central systems'.

NOTE 3: 'Groups of events' might, for example, be indicated in a statement of functional requirements at a high level of granularity by an input stream to an accounting software system labeled 'sales transactions'; or by an input stream to an avionics software system labeled 'pilot commands'.

With this definition, we can now define the following rules and a recommendation.

RULES - Levels of granularity for measuring a functional process

a) A functional size measurement of a piece of software requires that its FUR are known at the levels of granularity at which its functional processes and data movement sub-processes may be identified.

b) If some requirements must be measured before they have been defined in sufficient detail for a precise measurement, the requirements can be measured using an approximate approach. These approaches define how requirements can be measured at higher level(s) of granularity. Scaling factors are then applied to the measurements at the higher level(s) of granularity to produce an approximate size at the levels of granularity of the functional processes and their data movement sub processes. See the 'Guideline for Early or Rapid COSMIC Functional Size Measurement by approximation approaches'. [6].

In addition to the rules, COSMIC *recommends*[8] that the level of granularity at which functional processes and their data movement sub-processes are known should be the standard at which functional size measurements are required and used by providers of benchmarking services and of software tools designed to support or use functional size measurements, e.g. for estimating project effort.

BUSINESS EXAMPLE: The example, from the domain of business application software, is part of a well-known system for ordering goods over the Internet, which we will call the 'Everest Ordering Application'. The purpose of this example is to illustrate different levels of granularity and that functional processes may be revealed at different levels'. The description below is highly-simplified for the purposes of this illustration of levels of granularity.

---

[8] The reason that use of the functional process level of granularity is 'recommended' rather than given as a rule is that this recommendation applies not just to individual users of the COSMIC method but to their networks of suppliers of services and tools that use the size measurements. COSMIC can only make recommendations to this wider community.

If we wished to measure this application, we might assume the purpose of the measurement is to determine the functional size of the part of the application available to the human customer users (as 'functional users'). We would then define the scope of the measurement as 'the parts of the Everest application accessible to customers for ordering goods over the Internet'. Note, however, that the purpose of this example is to illustrate different levels of granularity. We will therefore explore only some parts of the system's total functionality sufficient to understand this concept of levels of granularity. This example is about levels of granularity of the FUR; it says nothing about any possible decomposition of the underlying software.

At the highest 'Level 1 (Main Function)' of this part of the application a statement of the requirements of the Everest Ordering Application would be a simple summary statement such as the following.

'The Everest Ordering Application must enable customers to enquire upon, select, order, pay for and obtain delivery of any item of Everest's product range, including products available from third party suppliers.'

Zooming-in on this highest-level statement of the requirements we find that at the next lower level 2 the Everest Ordering Application consists of four sub-functions, as shown on Figure 2.8 (a).

Level 1 (Main Function)

```
                        ┌─────────────┐
                        │   Everest    │
                        │  Ordering    │
                        │ Application  │
                        └──────┬──────┘
```

Level 2 (Sub-functions)

Enquiry/ Order	Checkout/ Payment	Order Follow-up	Account Maintenance
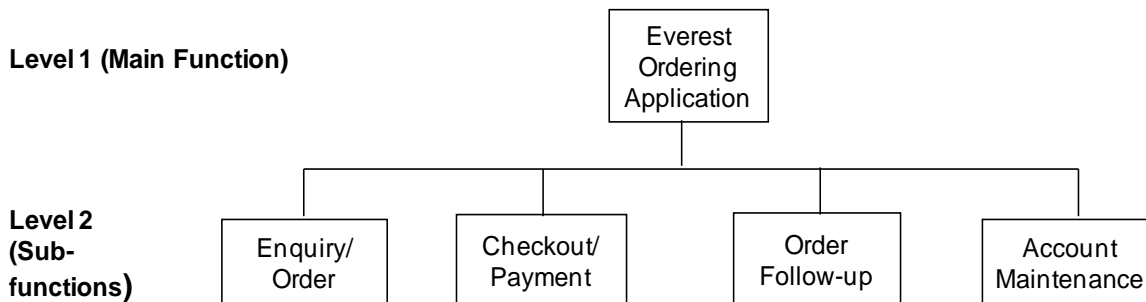
Figure 2.8 (a) - Analysis of the Everest Ordering System: the first two levels of granularity

The requirements of the four sub-functions are:

- *The Enquiry/Order sub-function which enables a customer to find any product in the Everest database, as well as its price and availability and to add any selected product to a 'basket' for purchase. This sub-function also promotes sales by suggesting special offers, offering reviews of selected items and enabling general enquiries such as on delivery terms, etc. It is a very complex sub-function. We therefore do not analyze this sub-function in any further detail below level 2 for the purposes of this example.*

- *The Checkout/Payment sub-function which enables a customer to commit to order and pay for the goods in the basket.*

- *The Order Follow-up sub-function which enables a customer to enquire how far an existing order has progressed in the delivery process, to maintain their order (e.g. change delivery address) and to return unsatisfactory goods.*

- *The Account Maintenance sub-function which enables an existing customer to maintain various details of his/her account such as home address, means of payment, etc.*

- *Figures 2.8 (b) and (c) show some detail that is revealed when we zoom-in on the requirements, down one further level of granularity on the Checkout/Payment sub-function, the Order Follow-up sub-function and the Account Maintenance sub-function. In this zooming-in process it is important to note that*

- *we have not changed the scope of the functionality to be measured, and*
- *all levels of the description of the Everest application show the functionality available to the customers (as functional users). A customer can 'see' the functionality of the application at all these levels of granularity.*

Level 2 (Sub-functions)

Checkout/ Payment Sub-function

Level 3 (All sub-sub-functions)

| Review, maintain confirm order | Select delivery/ Packaging options | Pay for order | Display & e-mail order confirmation |
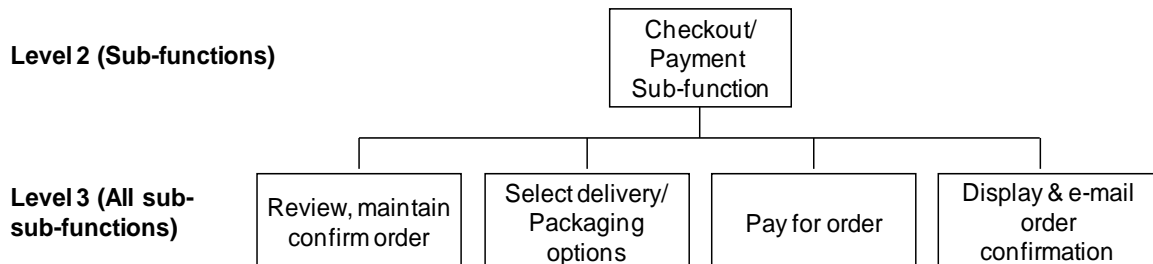
Figure 2.8 (b) - Analysis of the Checkout/Payment Sub-function

Level 2 (Sub-functions)

Order Follow-up Sub-function

Account Maintenance Sub-function

Level 3

| Enquire on current order Process | Enquire on historic orders Process | Returned goods Sub-sub-function | | Maintain customer details Sub-sub-function | Maintain payment method Sub-sub-function |
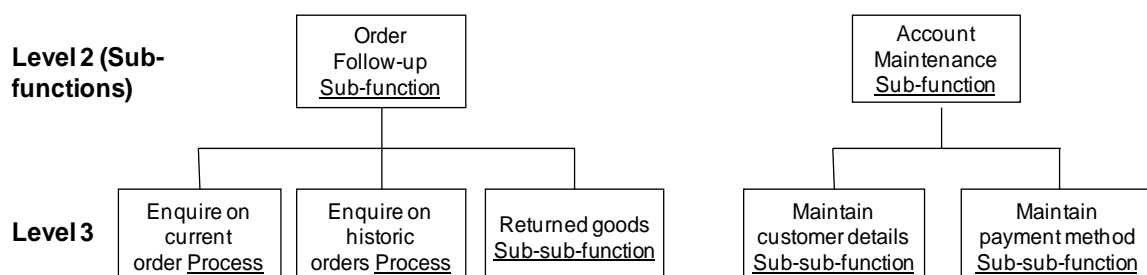
Figure 2.8 (c) - Analysis of the Order Follow-up and Account Maintenance Sub-function

Figure 2.8 (c) now reveals that when we zoom-in to the lower level 3 of this particular analysis of the Order Follow-up sub-function, we find two individual functional processes[9] at level 3 (for two enquiries of the Order follow-up sub-function). More functional processes would be revealed if we were to continue the refinement of the Level 3 sub-sub-functions to lower levels. This example demonstrates, therefore, that when some functionality is refined in a 'top-down' approach, it cannot be assumed that the functionality shown at a particular 'level' on a diagram will always correspond to the same 'level of granularity' as this concept is defined in the COSMIC method. (This definition requires that at any one level of granularity the functionality is 'at a comparable level of detail'.)

Furthermore, other analysts might well draw the diagrams differently, showing other groupings of functionality at each level of the diagram. There is not one 'correct' way of zooming in on the functionality of such a complex system.[10]

Given these variations that inevitably occur in practice, a Measurer must carefully examine the various levels of an analysis diagram to find the functional processes that must be measured. Where in practice this is not possible, for example because the analysis has not yet reached the level where all functional processes have been revealed, rule (b) above must be applied. To illustrate this, let us examine the case of the 'Maintain

---

[9] (Note: at this stage someone who is new to the COSMIC method may not be convinced on the basis of the definition and the rules for the 'functional process level of granularity' that the two Enquiries that have appeared at Level 3 are actually functional processes, as opposed to two sub-sub-systems that could be decomposed further. Section 3.2 on 'Identifying functional processes' will provide more evidence to support the analysis given here.)

[10] Figure 2.8 may not even be an example of best practice, but it is typical of how such diagrams may be drawn.

customer details sub-sub-function' (see Figure 2.8 (c) above), in the branch of the Account Maintenance sub-function.

To an experienced Measurer, the word 'maintain' almost invariably suggests a group of events and thus a group of functional processes. We can therefore assume that this 'Maintain' sub-sub-function must comprise three functional processes, namely an 'enquire on customer details', 'update customer details' and 'delete customer details'. (The 'create customer details' process must also obviously exist, but this occurs in another branch of the system, when a customer orders goods for the first time. It is outside the scope of this simplified example.)

An experienced Measurer should be able to 'guestimate' a size of this sub-sub-function in units of COSMIC Function Points by taking the assumed number of functional processes (three in this case) and multiplying this number by the average size of a functional process. This average size would be obtained by calibration in other parts of this system or in other comparable systems. Examples of this calibration process are given in the document ' Guideline on early or rapid COSMIC functional size measurement using approximation approaches' [6] which also contains other examples of other approaches to approximate sizing.

Clearly, such approximation methods have their limitations. If we apply such an approach to the Level 1 statement of requirements as given above ('The Everest application must enable customers to enquire upon, select, order, pay for and obtain delivery of any item of Everest's product range ….'), we could identify a few functional processes. But more detailed analysis would reveal that the real number of functional processes in this complex application must be much greater. That is why functional sizes usually appear to increase as more details of the requirements are established, even without changes in scope. These approximation methods must therefore be used with great care at high levels of granularity, when very little detail is available.

For an example of measuring at varying levels of granularity and of decomposition, see the telecoms system example in the 'Guideline for early or rapid COSMIC functional size measurement using approximation approaches' [6].

2.5   Concluding remarks on the Measurement Strategy Phase

It is essential to determine and document the parameters of the measurement strategy so as to ensure that the resulting size can be correctly understood and used in the future. The great majority of functional size measurements are carried out for a purpose that is related to development effort in some way, e.g. for project development performance measurement, or for project estimating. In many of these situations, a standard measurement pattern may be used: see the 'Guideline on Measurement Strategy Patterns' [5]).

3

THE MAPPING PHASE

3.0   Chapter summary

This chapter discusses the second, 'Mapping', phase of the measurement process by defining the key concepts of the Generic Software Model and the process to be followed in mapping the FUR of the software to the model, so that the FUR can be measured. These key concepts of the Generic Software Model are: (*in italics*):

- An event causes a *functional user* to request a service from the piece of software being measured. Such an event is called a '*triggering event'* and the requested service is called a '*functional process'*.

- Functional processes are composed of two types of sub-processes that either move data (*'data movements'*) or that manipulate data (*'data manipulation'*). Data manipulation sub-processes are not recognized separately but are considered to be accounted for by the data movements with which they are associated.

- A data movement moves a '*data group'*. A data group consists of *data attributes* that all describe one '*object of interest'*, i.e. a 'thing' that is of interest in the world of the functional user concerned.

- There are four types of data movements: *Entries* and *Exits* each move one data group in and out of a functional process across a *boundary* from/to a functional user respectively. *Reads* and *Writes* each move one data group between the functional process and *persistent storage*.

Each of these concepts is defined in this chapter, which includes comprehensive principles and rules to help identify the concepts correctly, and extensive examples of the concepts for different types of software.

3.1   Mapping the FUR to the Generic Software Model

Figure 3.0 shows the steps of the process for mapping the Functional User Requirements (FUR) as in the available software artefacts to the form required by the COSMIC Generic Software Model.

Each step in this process is the subject of a specific section indicated in the step's title bar in Figure 3.0.

The process is designed to be applicable to a very broad range of software artefacts. We encourage Measurers to use this general process to derive more specific rules for use in their local environment to map from the software artefacts produced by their local software engineering method to the COSMIC Generic Software Model. The aim of a specific local process, illustrated with local examples, should be to reduce uncertainty in the mapping and hence to improve the accuracy and repeatability of the measurements.
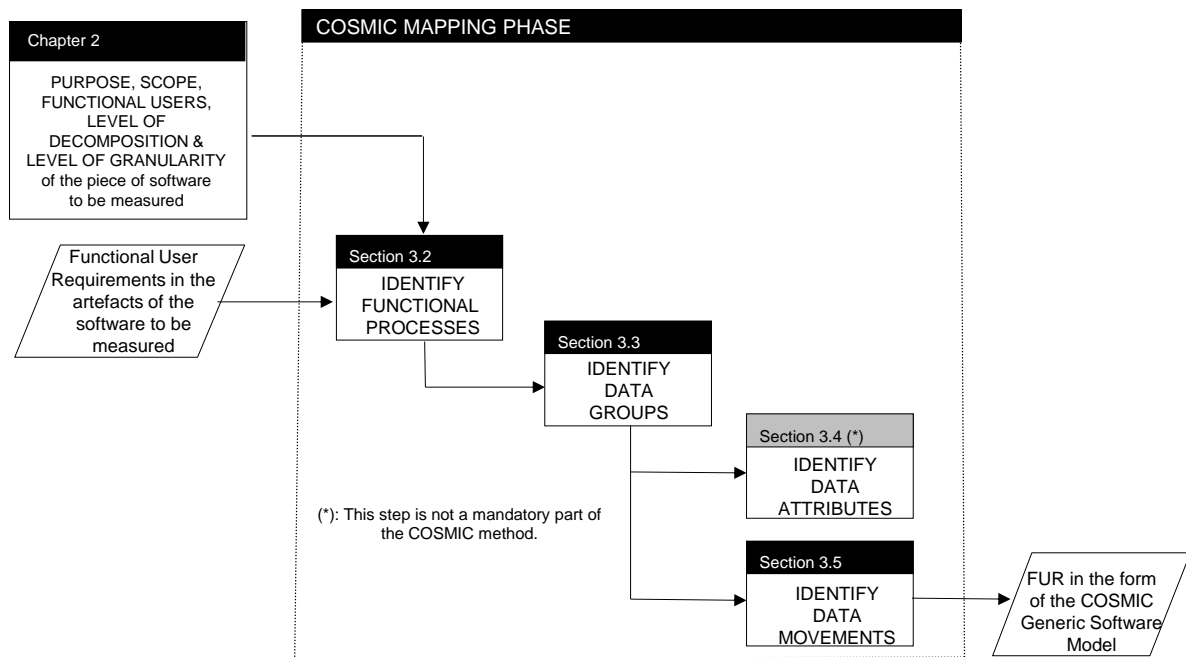
Figure 3.0 – General method of the COSMIC mapping process

Several guidelines are available that describe how to map from various data-analysis and requirements-determination methods, used in different domains to the concepts of the COSMIC method. Examples are the Guideline for Sizing Business Application Software [7], the Guideline for Sizing Data Warehouse Application Software [8], the Guideline for Sizing Service-Oriented Architecture Software [9] and the Guideline for Sizing Real-time Software [4]. For the business [10] and real-time domains [11] there are also Quick Reference Guides available that give an overview of the process in a few pages.

The aim of Figures 3.1 and 3.2 is to help our transition from the Software Context Model used in the Measurement Strategy phase to the Generic Software Model. The Figures apply for a piece of business application software and for a typical piece of real-time embedded software, and correspond to Figures 2.6 and 2.7 respectively.
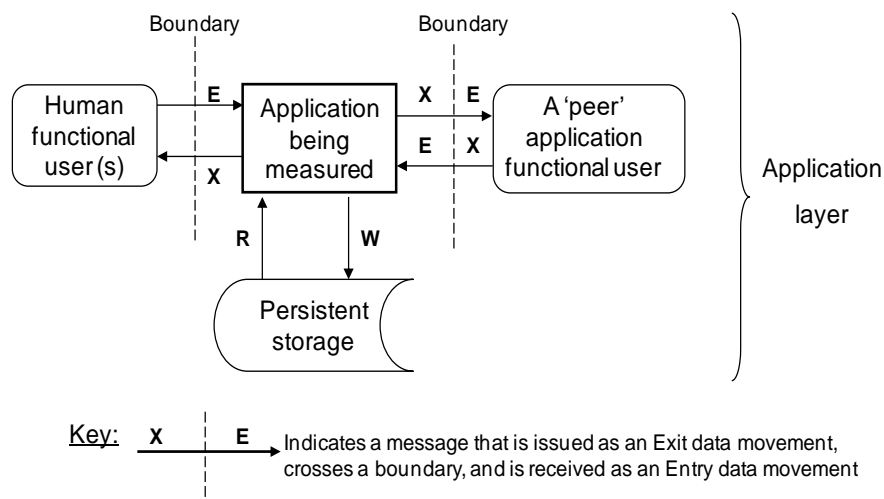
Figure 3.1 - A business application with humans and another 'peer' application as its functional users
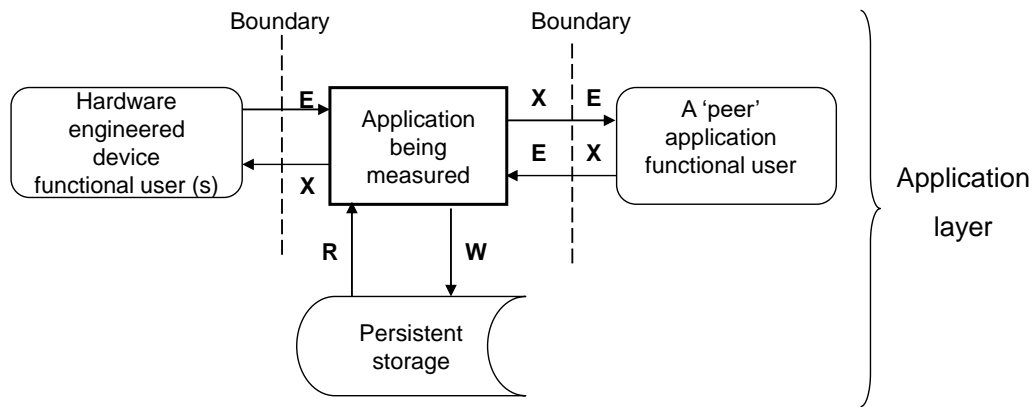
Figure 3.2 - A real-time embedded software application with various hardware engineered devices and peer software as its functional users

From principle f) of the Software Context Model (see section 1.3.1), we note that the functional users of the software being measured are the 'senders and/or or intended recipients of data to/from the software respectively'. Principle a) of the Generic Software Model (see section 1.3.2) tells us that a piece of software 'interacts with its functional users across a boundary and with persistent storage within this boundary'.

Fig. 3.1 shows the application software being measured has two functional users, human(s) and another peer application. The embedded software application of Figure 3.2 has only hardware devices and a peer application as its functional users. The arrows showing the movements of data are labeled with the abbreviations showing their types (E = Entry, X = eXit, R = Read, W = Write).

Note that 'persistent storage' refers to any logical storage that the software being measured is required to access via Read or Write data movements; it does not imply any type of physical storage.

3.2 Identifying functional processes

The first step of the Measurement Phase is to identify the set of functional processes of the piece of software to be measured, from its Functional User Requirements.

3.2.1 Definitions

DEFINITION – Event
Something that happens.

DEFINITION – Triggering event
An event, recognized in the Functional User Requirements of the software being measured, that causes one or more functional users of this software to each generate one or more data groups. The first data group generated by any one functional user will subsequently be moved by a triggering Entry. A triggering event cannot be sub-divided and has either happened or not happened.
NOTE: Clock and timing events can be triggering events.

DEFINITION – Functional process

a) A set of data movements, representing an elementary part of the Functional User Requirements for the software being measured, that is unique within these FUR and that can be defined independently of any other functional process in these FUR.

b) A functional process shall have only one triggering Entry. Each functional process starts processing on receipt of a data group moved by the triggering Entry data movement of the functional process.

c) The set of all data movements of a functional process is the set that is needed to meet its FUR for all the possible responses to its triggering Entry.

NOTE 1: When implemented, it is an *occurrence* of a functional process that starts *executing* on receipt of an *occurrence* of a data group moved by an *occurrence* of a triggering Entry.

NOTE 2: The FUR for a functional process may require one or more other Entries in addition to the triggering Entry.

NOTE 3: If a functional user sends a data group with errors, e.g. because a sensor-user is malfunctioning or data entered by a human has errors, it is usually the task of the functional process to determine if the event really occurred and/or if the entered data are really valid, and how to respond.

NOTE 4: A functional process is 'unique' (as in a) above), and its total size must be included in the size of the FUR, if it is initiated by a triggering Entry that results originally from a triggering event that is distinguished as unique within the FUR. Two or more functional processes within the same FUR may be unique, even though they share some common functionality. See section 3.2.7 for examples of functional processes with shared functionality.

DEFINITION – Triggering Entry

The Entry data movement of a functional process that moves a data group generated by a functional user that the functional process needs to start processing.

NOTE: This definition results from the Generic Software Model, which is a logical model. Physically, a functional process may start processing before data has been entered e.g. when a human user clicks on a menu to display a blank screen for data entry.

The relationship between a triggering event, the functional user and the Entry data movement that triggers a functional process being measured is depicted in Figure 3.3 below. The interpretation of this diagram is: *a triggering event causes a functional user to generate a data group that is moved by the triggering Entry of a functional process to start the functional process.*

NOTE: For ease of reading, we often omit the reference to the data group and state that a functional user *initiates* a triggering Entry that starts a functional process, or even more simply that a functional user *initiates* a functional process.
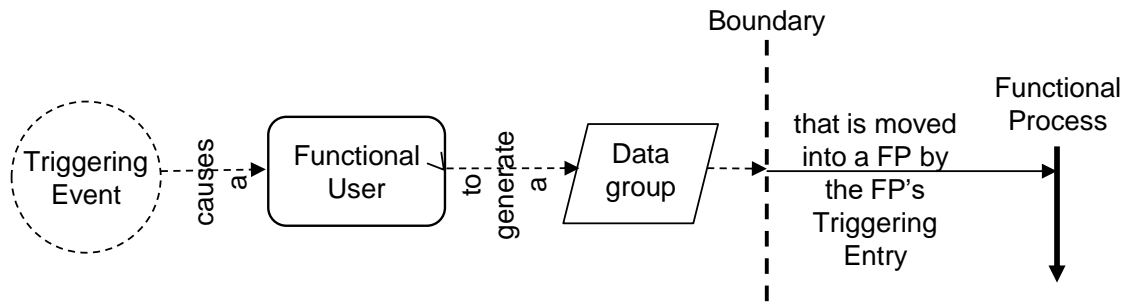
Figure 3.3 – Relationships between a triggering event, a functional user and a functional process

All the relationships between the concepts in Figure 3.3 (the triggering event / functional user / data group / triggering Entry / functional process) may be one-to-many, many-to-one, or many-to-many, with one exception. The exception is that the data group moved by any one triggering Entry may initiate only one functional process as per clause b) of the definition of a functional process. Some examples of possible cardinalities:

- A triggering event may be sensed by many functional users, e.g. an earthquake is detected by many sensors;

- Human functional users of a business application can detect many types of events. When a human user decides the event must be notified to the application, i.e. it is a triggering event, the user initiates a triggering Entry. (A human user may even effectively 'generate' an event when deciding to make an enquiry.) Similarly, a software application A that is a functional user of another application B (being measured), may 'call' application B for various different purposes. Each call (-type) corresponds to application A generating a separate triggering event.

- A hardware functional user may initiate more than one triggering Entry on sensing one triggering event in certain types of safety-critical software;

- Some functional processes may be initiated by different functional users, e.g. a re-usable software component that can be called by any of its software functional users.

In practice, the cardinalities along all the chains for the software being measured (i.e. that describe the specific events that cause specific functional users to initiate specific functional processes) will be constrained by the software FUR. For a fuller discussion of the cardinalities along the chain of Figure 3.3 and for more examples, see Appendix C to this Measurement Manual.

The data group(s) that a functional user will generate as a result of a triggering event depend on the FUR of the functional process that will process the data, as illustrated by the following examples.

BUSINESS EXAMPLE 1: A functional process of a personnel software system may be started by the triggering Entry that moves a data group describing a new employee. The data group is generated by a human functional user of the personnel software who enters the data.

REAL-TIME EXAMPLE 2: A functional process of a real-time software system may be started by its triggering Entry informing the functional process that a clock (functional user) has ticked. The data group moved conveys data (the tick, perhaps via a single bit) that informs only that an event has occurred.

REAL-TIME EXAMPLE 3: A functional process of an industrial real-time fire detection software system may be started by its triggering Entry initiated by a specific smoke detector (functional user). The data group generated by the detector conveys the

information 'smoke detected' (an event has occurred) and includes the detector ID (i.e. data that can be used to determine where the event occurred).

REAL-TIME EXAMPLE 4: A bar code reader (a functional user) at a supermarket checkout starts a scan when a bar code appears in its window (the triggering event). The reader generates a data group, comprising an image of the bar code that is input to the checkout software. The data group image is moved by a triggering Entry into its functional process. The latter adds the product cost to the customer's bill if the code is valid, sounds a 'beep' to inform the customer that the product has been accepted, and logs the sale etc., etc.

3.2.2 The approach to identifying functional processes

The approach to identifying functional processes depends on the software artefacts that are available to the Measurer, which in turn depend on the point in the software lifecycle when the measurement is required and on the software analysis, design and development methods in use. Since the latter vary enormously, this Measurement Manual can only provide a general process for identifying functional processes.

The process of identifying functional processes, after the functional users have been identified, given the FUR for the software being measured and considering the examples in Appendix C, follows the chain of Figure 3.3, namely:

- Identify the separate events in the world of the functional users that the software being measured must respond to – the 'triggering events' (Triggering events can be identified in state diagrams and in entity life-cycle diagrams, since some state transitions and entity life-cycle transitions correspond to triggering events to which the software must react.);

- Identify which functional user(s) of the software may respond to each triggering event;

- Identify the triggering Entry (or Entries) that each functional user may initiate in response to the event;

- Identify the functional process started by each triggering Entry.

Use the following rules to check that candidate functional processes have been properly identified.

RULES – Functional process

a) A functional process shall belong entirely to the measurement scope of one piece of software in one, and only one, layer.

b) A functional process shall comprise a minimum of two data movements, namely the triggering Entry plus either an Exit or a Write, giving a minimum size of 2 CFP. There is no upper limit to the number of data movements in a functional process and hence no upper limit to its size.

c) An executing functional process shall be considered terminated when it has satisfied its FUR for all the possible responses to its triggering Entry. A pause during the processing for technical reasons shall not be considered as termination of the functional process.

3.2.3 Triggering events and functional processes in the business applications domain

a) Triggering events of an on-line business application usually occur in the real world of the human functional users of the application. The human user conveys the occurrence of the event to a functional process by entering data about the event.

BUSINESS EXAMPLE 1:   In a company, an order is received (triggering event), causing an employee (functional user) to enter the order data (triggering Entry conveying data about the object of interest 'order'), as the first data movement of the 'order entry' functional process.

b) Separate triggering event (-types) and therefore separate functional process (-types) should be distinguished in the following cases:

When a human functional user makes decisions outside the software on 'what to do next' that are independent in time and that require separate responses from the software, each separate decision is a separate triggering event for which the software must provide a separate functional process.

BUSINESS EXAMPLE 2: A functional user enters a customer order for an item of complex industrial equipment and later confirms acceptance of the order to the customer. Between entering the order and accepting it, the user may make enquiries about whether the new order can be delivered by the requested delivery date, and about the customer's credit-worthiness, etc. Although acceptance of an order must follow entry of an order, in this case the user must make a separate decision to accept the order. This indicates separate functional processes for order entry and for order acceptance (and for each of the enquiries).

When the responsibilities for activities are separated.

BUSINESS EXAMPLE 3: In a personnel system where the responsibility for maintaining basic personal data is separated from the responsibility for maintaining payroll data indicating separate functional users each with their own separate functional processes.

c) An application A has a peer application B that needs to send data to, or obtain data from, application A. Application B triggers a functional process in application A when it needs to send data to or obtain data from application A. Application B is then a functional user of the application A.

BUSINESS EXAMPLE 4: Suppose that on receipt of an order in Business Example 1, the order-processing application is required to send the details of any new client to a central client-registration application, which is being measured. The order-processing application is thus a functional user of the central application. The order-processing application, having received data about a new client, generates the client data group which it sends to the central client-registration application which triggers a functional process to store these data.

d) There is no difference in principle to the analysis of a functional process whether it is required to be processed on-line or in batch mode. By definition, all input data for batch processing must either have been temporarily stored somewhere before the process can start, or it may be transmitted in as a batch stream. See Figure 3.4. (N.B. we distinguish input data - all the Entries - from persistent data that might also need to be read or written by the batch process.) When measuring batch-processed software, any temporarily-stored input data or input data stream should be analysed in the same way as if it were being entered directly to the application. This input data is not 'persistent data'.

All the Entries

Data transmitted in

Input data stored temporarily

One or more functional processes implemented in a batch-processed program (which is started by e.g. the OS or a human operator)

All the Exits

Output **data** e.g. in reports

File(s) to be created, read and maintained, i.e. the persistent data
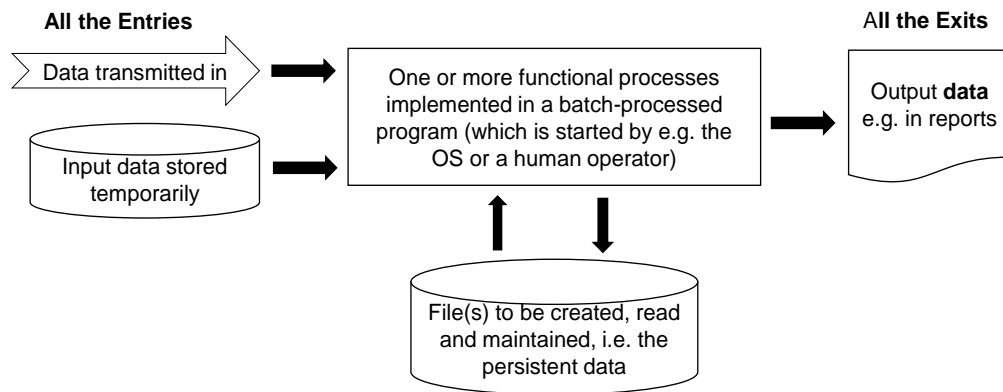
Figure 3.4 - A batch-processed 'job', implementing a collection of functional processes

NOTE: A requirement that some input data be batch-processed is a non-functional requirement (NFR). The effect of this NFR is that the input data must be available (as a 'batch') for input to the batch processed application. How that happens in practice does not concern the analysis of the FUR of the batch-processed software.

Note also that each functional process (-type) to be processed in a batch 'job' should be analyzed in its entirety, independently of any other functional processes in the same job. Each functional process in the job must have its own triggering Entry.

BUSINESS EXAMPLE 5: Suppose the orders in the Business Example 1 above are entered by an 'off-line' process in some way, e.g. by optical scanning of paper documents and are stored temporarily for automatic batch processing. How is this order-entry functional process analyzed if it is to be batch-processed? The functional user is the human who causes the order data to be entered off-line ready to be processed in batch mode; the triggering Entry of the functional process that will process the orders in batch mode is the data movement that moves the order data group from temporary storage into the process. (The off-line process, if it must be measured, involves another, separate functional process. Effectively, the functional user has initiated two triggering Entries, one starts the off-line process to load the orders to temporary storage and the other to start the batch processing of the orders.)

BUSINESS EXAMPLE 6: Suppose a FUR for an end-of year batch-processed application is to report the outcome of business for the year, and to reset positions for the start of the next year. Physically, an end-of-year clock tick generated by the operating system causes the application to start processing. Logically, however, each functional process of the application takes its input data from the stream of data to be batch-processed. This should be analyzed in the normal way (e.g. the input data for any one functional process comprises one or more Entries, the first of which is the triggering Entry for that process).

However, assume there is a particular functional process of the batch-processed application that does not require any input data to produce its set of reports. Physically, the (human) functional user has delegated to the operating system the task of triggering this functional process. Since every functional process must have a triggering Entry, we may consider the end-of-year clock tick that started the batch stream as filling this role for this process. This functional process may then need several Reads and many Exits to produce its reports. Logically, the analysis of this example is no different if the human functional user initiates the production of one or more reports via a mouse click on an on-line menu item, rather than delegating the triggering of the report production in batch mode to the operating system.

For several examples on distinguishing triggering events and functional processes in batch streams, see the 'Guideline for Sizing Business Application Software using COSMIC' [7], section 4.6.3.

3.2.4 Triggering events and functional processes in the real-time applications domain

a) A triggering event is typically detected by a sensor.

REAL-TIME EXAMPLE 1: When a sensor (functional user) detects that the temperature reaches a certain value (triggering event), the sensor sends a signal to initiate a triggering Entry data movement of a functional process to switch off a heater (another functional user).

REAL-TIME EXAMPLE 2: A military aircraft has a sensor that detects the event 'missile approaching'. The sensor is a functional user of the software that must respond to the threat. For this software, an event occurs only when the sensor detects something, and it is the sensor (the functional user) that generates a data group to initiate a triggering Entry saying, e.g. 'sensor 2 has detected a missile', plus maybe a stream of data about how fast the missile is approaching and its co-ordinates.

b) Periodic signals from a clock ('clock ticks') can trigger a functional process.

See Real-time Example 2 in section 3.2.1. There is no other data accompanying the clock-tick. The functional process obtains data from various sensors and takes whatever action is needed.

3.2.5 More on separate functional processes

Software distinguishes events and provides the corresponding functional processes depending only on its FUR. When sizing software, it can sometimes be difficult to decide what the separate events are that the software is required to recognize. This is especially the case where the original requirements are no longer available or where, for example the developer may have found it economical to implement several requirements in one physical transaction. It may help with the analysis to examine the organization of data input (see below) or to examine the menus for some installed software to help distinguish the separate events that the software must respond to and the corresponding functional processes.

BUSINESS EXAMPLE 1: Suppose there is a functional user requirement for two types of social benefits, first for an additional child and second a 'working tax credit' for those on low income. These are requirements for the software to respond to two events that are separate in the world of the human functional users. Hence there should be two functional processes, even though a single tax form may have been used to capture data for both cases.

According to clause c) of its definition, a functional process must 'meet its FUR for all the possible responses to its triggering Entry'. This means that the same one functional process *type* must be able to deal with all possible *occurrences* of *values* of the data attributes of the data group moved by its triggering Entry, including both valid and invalid data values, and even in some cases missing data values. All these variations in values of the data moved by the triggering Entry will usually result in different processing paths being followed when the functional process executes. But in spite of all these variations, we must still identify only the one functional process type started by its one triggering Entry type. (The Measurer only needs to identify all the data movements of this functional process; the various processing paths in which they may occur are irrelevant to the measurement.)

BUSINESS EXAMPLE 2: A functional process that provides a general search capability against a database may be required to accept up to four search parameters (attributes of

its triggering Entry). But the same functional process will function if the values of only one, two or three search parameters are entered.

BUSINESS EXAMPLE 3: For a functional process to register a new customer for a car rental company, it is mandatory to enter data for most data attributes, but some (e.g. some contact details) are optional and may be left blank. Regardless of whether all or a sub-set of these attributes are entered there is only one functional process for registering a new customer.

BUSINESS EXAMPLE 4: Continuing from Example 3, for the functional process to make a car rental reservation in the same company, there are several options which may or may not be taken up, e.g. for extra insurance, additional drivers, requests for child seats, etc. These different options lead to different processing paths within the car rental reservation functional process, but there is still only one functional process for reserving a car rental.

REAL-TIME EXAMPLE: One triggering Entry (aircraft altitude information sent by the Geographical Positioning System) to a functional process of an avionics system will trigger one of two quite different processing paths within the functional process depending on the value of the Entry, i.e. whether the altitude is above or below a given height. The different paths will display different data groups on the pilot's map and, if the altitude is too low, additional warnings will be issued. There is only one functional process.

Once identified, each functional process can be registered on an individual line, under the appropriate layer and named piece of software, in the Generic Software Model matrix (Appendix A).

3.2.6 Measuring the components of a distributed software system

When the purpose of a measurement is to measure separately the size of each component of a distributed software system, a separate measurement scope must be defined for each component. In such a case the sizing of the functional processes of each component follows all the normal rules as already described.

From the process for each measurement (… define the scope, then the functional users and boundary, etc. …) it follows that if a piece of software consists of two or more components, there cannot be any overlap between the measurement scope of each component. The measurement scope for each component must define a set of complete functional processes. For example, there cannot be a functional process with part in one scope and part in another. Likewise, the functional processes within the measurement scope for one component do not have any information about the functional processes within the scope of another component, even though the two components exchange messages.

The functional user(s) of each component is/are determined by examining where the events occur that trigger functional processes in the component being examined. (Triggering events can only occur in the world of a functional user.)

Figure 3.7 illustrates the functional processes of the two components of a client-server distributed system and the data movements that they exchange.

3.2.7 Independence of functional processes sharing some common or similar functionality: re-use

Any two or more functional processes in the same software being measured may have some functionality that is identical or very similar in each process. This phenomenon is referred to as 'functional commonality', or functional 'similarity'.

However, in the COSMIC method (as in all other FSM Methods) each functional process is defined, modelled and measured independently of, i.e. without reference to, any other functional process in the same software being measured (see clause a) of the definition of a functional process). Any required functionality that is common to or similar across any two or

more functional processes in the same software being measured must be included when measuring the size for each of these functional processes. The following are examples of functional commonality or similarity that may be found in practice.

BUSINESS EXAMPLE: Several functional processes in the same software being measured may need the same validation functionality for e.g. 'date of order', or may need to access the same persistent data, or may need to carry out the same interest calculation.

REAL-TIME EXAMPLE: Several functional processes in the same software being measured may need to obtain data from the same sensor (common movement of same data group) or may need to carry out the same scale conversion calculation, e.g. from Fahrenheit to Centigrade (common data manipulation).

When a statement of FUR is implemented in software, any 'functional commonality' may or may not be developed as re-usable software. All implementation decisions including the extent of actual or potential software re-use must be ignored when measuring functional size. However, re-use may need to be taken into account when using functional size measurements for project effort analysis or estimating purposes.

3.2.8   Events that trigger a software system to start executing

When measuring the size of a piece of software, identify only the events and corresponding triggering Entries that trigger the functional processes that the software must respond to as defined in its FUR. Functionality needed to start-up (or 'launch') the software itself is not part of these functional processes and should be ignored (or measured separately, if required). The following examples describe how software is started in three domains.

BUSINESS EXAMPLE: For a business application, the functional user that starts the application may be a scheduler component of the operating system, a computer operator, or any other human user (e.g. when a PC user launches a browser or word-processing software).

REAL-TIME EXAMPLE: For a real-time application, the functional user that starts the application may be the operating system or network management generating a clock signal, or a human operator (e.g. to start a process control system from an operator workstation).

INFRASTRUCTURE EXAMPLE: For a computer operating system, the functional user that starts the operating system is a bootstrap program that is started when the computer power is switched on.

The following are examples from two domains of the relationships that may exist, if any, between the event/process that starts the software to be measured and the events/processes that the software must execute as described in its FUR.

BUSINESS EXAMPLE 1: An application to process the input data for a variety of functional processes in batch mode may be started by a scheduler of the operating system. If the purpose is to measure the FUR of the batch application, the 'start-the-system' functionality should be ignored. The triggering Entries for the functional processes of the batch-processed application and any other Entries that may be required will form the input data for the batch application.

BUSINESS EXAMPLE 2: Exceptionally, a batch-processed application to produce summary reports at the end of a time-period may be started without needing any input data provided directly from the functional user. For the analysis see Business Example 6 in section 3.2.3.

REAL-TIME EXAMPLE: A modern vehicle has a distributed system of Electronic Control Units (ECUs) to control many functions, e.g. engine management, brakes, air-

conditioning, etc. In the AUTOSAR architecture, in a distributed system, the 'Network Management' (NM) module, which is always running, is responsible for activating the ECUs that are connected together via a network ('bus'). This NM module also handles the coordinated switching between the ECU operating states: Normal Operation, Low Power and Sleep. Therefore it is the NM that wakes up or puts to sleep ECUs. When measuring any ECU application software, this NM functionality should be ignored.

3.3   Identifying objects of interest and data groups

3.3.1   Definitions and principles

Having identified the functional processes, the next main aim must be to identify their data movements. To do this we must recall the Generic Software Model (see section 1.3.2) and the introduction to the concepts of this chapter in section 3.0, which state that a data movement 'moves a *data group*, whose *data attributes* all describe a single *object of interest'.* Therefore, to understand a data movement, we must first define and understand these three concepts.

DEFINITION – Object of interest
Any 'thing' in the world of the functional user that is identified in the Functional User Requirements of the software being measured, about which the software is required to move a data group in or out of the software, or to or from persistent storage. It may be any physical thing, as well as any conceptual object or part of a conceptual object.
NOTE 1: In the COSMIC method, the term 'object of interest' is used in order to avoid terms related to specific software engineering methods. The term does not imply 'objects' in the sense used in Object Oriented methods.
NOTE 2: When a functional user sends a data group describing itself, e.g. its state or its identity, or when a functional user receives data describing itself, then the functional user also fulfils the role of the 'thing' in its world, so it is also the object of interest described by the data group.
NOTE 3: There is nothing absolute about an object of interest. A 'thing' may be an object 'of interest' to a functional user via one or more functional processes, but not be an object 'of interest' to another functional user via other functional processes, even in the same software being measured.

For more on Note 2 of this definition, see section 3.3.4.

DEFINITION – Data group
A distinct, non-empty and non-ordered set of data attributes where each included data attribute describes a complementary aspect of the same one object of interest.
NOTE: The term 'data group' does not necessarily mean 'the set of *all* data attributes that describe a single object of interest'. The FUR of a piece of software may specify data groups to be formed from any combinations of data attributes that all describe the same object of interest, as needed by different functional processes.

Once identified, each candidate data group must comply with the following principle:

PRINCIPLE – Data group
Each identified data group shall be unique and distinguishable through its unique collection of data attributes.

DEFINITION – Data attribute
The smallest parcel of information, within an identified data group, carrying a meaning from the perspective of the software's Functional User Requirements.
NOTE: Synonym 'Data Element'

(For more on Data Attributes, see section 3.4.)

In practice, the materialization of a data group can take many forms, e.g.:

a) As a physical record structure on a hardware storage device (file, database table, ROM memory, etc.).

b) As a physical structure within the computer's volatile memory (data structure allocated dynamically or through a pre-allocated block of memory space).

c) As the clustered presentation of functionally-related data attributes on an input/output device (display screen, printed report, control panel display, etc.).

d) As a message in transmission between a device and a computer, or over a network, etc.

Once identified, each data group can be registered in an individual column in the Generic Software Model matrix (Appendix A), under the 'Data Group Name' heading.

3.3.2 About the identification of objects of interest and data groups

The definition and principles of objects of interest and of data groups are intentionally broad in order to be applicable to the widest possible range of software. This quality sometimes results in it being difficult to apply the definition and principles when measuring a specific piece of software. Therefore, the following examples are designed to assist in the application of the principles to specific cases.

When faced with a need to analyze a group of data attributes that is moved in or out of a functional process or is moved by a functional process to or from persistent storage, *it is critically important* to decide if the attributes all convey data about a single 'object of interest', since it is the latter that determine the number of separate 'data groups' as defined by the COSMIC method that will be moved by data movements. For instance, if the data attributes to be input to a functional process are attributes of three separate objects of interest, then we need to identify three separate 'Entry' data movements.

This issue of deciding on the number of data groups can be especially difficult when analyzing the output of a functional process of a business application which may include:

- multiple data groups, each describing a different object of interest, e.g. a report showing totals at various levels of aggregation;

- the results of enquiries where the output will vary depending on the input;

- data groups that may even be unrelated to each other, e.g. an invoice which includes an advertisement for an unrelated service.

The following rule aims to assist the identification of data groups and hence objects of interest particularly in the output of functional processes. However, the rule applies equally to the input to a functional process as well as to its movements to and from persistent storage and to the FUR of software in any functional domain.

RULE - Identifying different data groups (and hence different objects of interest) moved in the same one functional process
For all the data attributes appearing in the input of a functional process:
a) sets of data attributes that have different frequencies of occurrence describe different objects of interest;
b) sets of data attributes that have the same frequency of occurrence but different identifying key attribute(s) describe different objects of interest;
c) all the data attributes in a set resulting from applying parts a) and b) of this rule belong to the same one data group type, unless the FUR specify that there may be more than one data group type describing the same object of interest in the input to the functional process (see Note 3)
This same rule applies for all the data attributes appearing in the output of a functional process, or all that are moved from a functional process to persistent storage, or all that are moved from persistent storage into a functional process.
NOTE 1. It can be helpful when analyzing complex output, e.g. reports with data describing several objects of interest, to consider each separate candidate data group as if it were output by one separate functional process. Each of the data group types identified this way must also be distinguished and counted when measuring the complex report. For examples, see the 'Guideline for sizing business application software' [7], in particular the example in section 2.6.1 and its analysis in 2.6.2. See also the analysis of the examples 4 and 5 in section 4.2.4.
NOTE 2. Examining how data attributes are physically grouped or separated on *input or output* may help distinguish different data group types, but cannot be relied upon to distinguish them. As an example, two or more sets of data attributes occurring on the same input or output that are physically separated for aesthetic reasons or for ease of understanding, will belong to the same one data group type if they satisfy the rule above.
NOTE 3. See section 3.5 of the Measurement Manual for the definitions, principles and rules for the *data movements* that move data groups, and section 3.5.7 (Examples 2, 3, 4 and 5) and 3.5.11 for exceptions to these rules for data movements, as per rule c above.

See Business Example 3 in section 3.3.2 as a simple illustration of the application of this rule. For more complex examples, see the Guideline for sizing Business Application software', [7].

Objects of interest and data groups in the business applications domain

In business applications, data is 'persistent' if it remains stored beyond the life of a functional process, or 'transient' if it exists only in the input or output of a functional process.

Note: the distinction between persistent or transient does not apply to objects of interest; they are 'things', physical or conceptual, identified in the FUR of the software to be measured.

BUSINESS EXAMPLE 1: In the domain of business application software, an object of interest could be 'employee' (physical) or 'order' (conceptual). The data groups describing these objects of interest would be made persistent if the software is required to store data about employees or orders. In the case of 'order', it commonly follows from the FUR of multi-line orders that two objects of interest are identified: 'order' and 'order-line'. The corresponding data groups could be named 'order data', and 'order-line data'.

Transient data groups are formed whenever there is an enquiry (or report-request) which asks for data about one or more 'things', i.e. 'objects *of interest'*, about which data is not held on persistent storage, but which can be derived from data held on persistent storage or derived just by data manipulation. The input data group of such an enquiry (the selection parameters to derive the required data) and the output data group(s) (containing the desired attributes) are both transient data groups because they exist only in the input and output respectively and do not survive the execution of the functional process.

BUSINESS EXAMPLE 2: Assume a FUR for an ad hoc enquiry functional process against a personnel database to find out the number of employees older than a given age, which must be input. The input parameter (the age limit) is a transient data group that defines an object of interest 'the set of employees aged over the given limit'. The output data group, comprising the count of employees aged over the given limit, is also transient and describes the same object of interest as the input. The data from which the count is derived (the employee file) is persistent.

BUSINESS EXAMPLE 3: Assume the same ad hoc enquiry against a personnel database as in Business Example 2, but in addition to outputting the total number of employees over the given age limit, the enquiry must also list the names of all employees aged over the given limit. The analysis is as for Business Example 2 but the functional process must now output two data groups: the count of employees (transient data) and the list of names of employees over the given age limit (derived from persistent data). These must be two separate data groups because they have different frequencies of occurrence (one count of employees for zero, one or more employee names (see the rule above),

An important general lesson from Business Example 3 is that a 'set' and a 'member of a set' are always different 'things' so must be different objects of interest. In the case of this example, the functional process outputs data groups describing two objects of interest, namely the 'set of employees older than the given age limit', and 'employee'. Their single-attribute data groups are 'count of employees over the given age limit' and 'employee name', respectively.

For a detailed discussion on methods of analyzing data to determine objects of interest and separate data groups, the reader is referred to the 'Guideline for Sizing Business Application Software using COSMIC' [7].

Objects of interest and data groups in the real-time software domain

In real-time software, there is usually no need to think about data as transient. In general, all data moving in or out of a real-time functional process is transient unless it is made persistent (e.g. for logging purposes), or has been obtained from persistent storage. The following are examples of data groups and corresponding objects of interest in real-time software.

REAL-TIME EXAMPLE 4: A data group entering software from a physical device may inform:

- *about the current state of an object of interest, such as that a valve is open or closed. (See section 3.3.4 on when a functional user sends data about itself, so is also the object of interest.)*

- *that a triggering event has occurred, leading to the start of a functional process (that may interrupt an already-executing functional process). The event is the object of interest.*

Similarly a data group output to a physical device, such as to switch a warning lamp on or off conveys data about the lamp object of interest.

REAL-TIME EXAMPLE 5: A message-switch software system may receive a message data group as input and route it forward unchanged as output, as per the FUR of the particular piece of software. The attributes of the message data group could be, for example, 'message ID, sender ID, recipient ID, route_code and message_content'; the object of interest of the message is 'message'.

REAL-TIME EXAMPLE 6: A common data structure represents objects of interest that are mentioned in the FUR, which can be maintained by functional processes, and which is accessible to most of the functional processes found in the measured software.

REAL-TIME EXAMPLE 7: A reference data structure, represents objects of interest whose attribute-values are given in graphs or tables found in the FUR, and which are held in permanent memory (ROM memory, for instance) and accessible to most of the functional processes found in the measured software.

REAL-TIME EXAMPLE: 8: Files, commonly designated as 'flat files', represent objects of interest mentioned in the FUR, which are held on a persistent storage device.

3.3.3 Data or groups of data that are not candidates for data movements

Any data appearing on input or output screens or reports that are not related to an object of interest to a functional user should <u>not</u> be identified as indicating a data group, so should not be measured.

BUSINESS EXAMPLE 1: Application-general data such as headers and footers (company name, application name, system date, etc.) appearing on all screens.

BUSINESS EXAMPLE 2: 'Control commands' (a concept defined only in the business application domain) that enables a functional user to control their use of the software, rather than to move data, e.g. page up/down commands, clicking 'OK' to acknowledge an error message, etc., see further section 3.5.10.

The COSMIC Generic Software Model assumes that all manipulation of data within a functional process is associated with the four data movement types, see section 3.5.6. Hence no data groups may be identified arising from data manipulation within a functional process in addition to the data groups moved by the Entries, Exits, Reads and Writes of the process. (For examples of manipulation and movements of data that might be mis-interpreted as data movements, see section 3.5.4, principle c) for a Read, and section 3.5.5 principle d) for a Write).

3.3.4 The functional user as object of interest

As pointed out in NOTE 2 to the definition of an object of interest, a functional user of the software being measured may also fulfil the role of the object or interest of a data group sent or received by the functional user.

In many real-time software systems, such as described in Real-time Example 1 in section 3.3.2 above, the Strategy phase may show that a physical device such as a sensor could be a functional user. Later, in the Mapping phase, this same sensor may also be identified as an object of interest. This is simply because the physical device 'interacts with the software' and at the same time the physical device is a 'thing...about which the software is required to move a data group in or out of the software, or to or from persistent storage'. In effect the physical device interacts with the software across the boundary to provide or to receive information about itself in the form of a data group describing some aspect of itself.

Similarly in business application software, a human functional user can also be the object of interest of the data group entered or received by the human user.

BUSINESS EXAMPLE: A human functional user enters an ID and a password in a logon process to identify himself/herself to a system. The object of interest of the data group entered is the human user.

REAL-TIME EXAMPLE: Suppose a temperature sensor 'A' sends a measure of the current temperature of a material for processing by a functional process. In one view the sensor can be regarded as providing information about its own state. The sensor therefore meets the criteria for identification as an object of interest and may be mapped from the FUR as such. However, the physical sensor is in the world of the users and '...interacts with the software...across the boundary', it therefore also meets the criteria for identification as a functional user and so must appear on the software context diagram.

3.4   Identifying data attributes (optional)

This section discusses the identification of data attributes referenced by the piece of software to be measured. In the COSMIC method, it is not mandatory to identify the data attributes. However, understanding the concept of a 'data attribute' is necessary in order to understand the section on 'measuring changes...', where a FUR to change a data attribute can result in the data movement to which the attribute belongs being measured. Also, it may be helpful to analyze and identify data attributes in the process of distinguishing data groups and objects of interest. Data attributes might also be identified if a sub-unit of the size measure is required, as presented in section 4.5 'Extending the COSMIC measurement method'.

3.4.1   Data attribute examples

(For the definition of a Data Attribute, see section 3.3.1.)

BUSINESS EXAMPLE 1: An 'employee' object of interest may be described by a data group called 'Employee master data', which contains the data attributes 'Employee ID', 'Name', 'Address', 'Date of birth', 'Sex', 'Marital status', 'National Insurance number', 'Grade', 'Job title', etc.

REAL-TIME EXAMPLES 2:   A thermocouple may, on request, report the attribute 'Temperature', A sensor of a security system may detect an intruder and send the attribute 'Movement detected'. A message in transmission may consist of the attributes 'From (address), To (address), Contents'.

3.4.2   About the association of data attributes and data groups

In theory, a data group might contain only one data attribute if this is all that is required, from the perspective of the Functional User Requirements, to describe the object of interest. In practice, such cases occur commonly in real-time application software (e.g. the data group entered to convey the tick of a real-time clock or the entry of the state of a sensor); they are less common in business application software.

3.5   Identifying the data movements

This step consists in identifying the data movements (Entry, Exit, Read and Write) of each functional process.

3.5.1   Definition of the data movement types

DEFINITION – Data movement
A base functional component which moves a single data group type.
NOTE 1: There are four sub-types of a data movement (-type) namely: Entry, Exit, Read and Write (-types).

NOTE 2: For measurement purposes, each data movement is considered to account for certain associated data manipulations – see section 3.5.6 for details.

NOTE 3: More precisely, it is an *occurrence* of a data movement, not a data movement type that actually moves the data group *occurrences* (not types). This comment also applies to the definitions of Entry, Exit, Read and Write.

DEFINITION – Entry (E)

A data movement that moves a data group from a functional user across the boundary into the functional process where it is required.

NOTE: An Entry is considered to account for certain associated data manipulations (see section 3.5.6).

DEFINITION – Exit (X)

A data movement that moves a data group from a functional process across the boundary to the functional user that requires it.

NOTE: An Exit is considered to account for certain associated data manipulations (see section 3.5.6).

DEFINITION – Read (R)

A data movement that moves a data group from persistent storage into the functional process which requires it.

NOTE: A Read is considered to account for certain associated data manipulation (see section 3.5.6).

DEFINITION – Write (W)

A data movement that moves a data group from inside a functional process to persistent storage.

NOTE A Write is considered to account for certain associated data manipulation (see section 3.5.6).

Figure 3.5, below, illustrates the overall relationship between the four types of data movement, the functional process to which they belong and the boundary of the measured software.
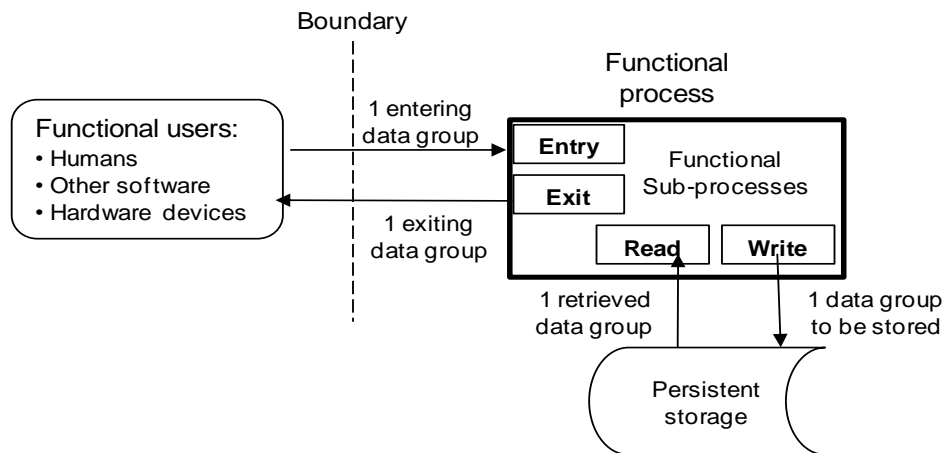
Figure 3.5 – The four types of data movement, each moving one data group, and their relationship with a functional process. (A functional process can, of course have many E, X, R and W data movements.)

3.5.2   Identifying Entries (E)

A candidate Entry data movement must comply with the following principles:

PRINCIPLES – Entry (E)
a) An Entry shall move a single data group describing a single object of interest from a functional user across the boundary and into the functional process of which the Entry forms part. If the input to a functional process comprises more than one data group, each describing a different object of interest, identify one Entry for each unique data group in the input. (See also section 3.5.7 on 'Data movement uniqueness'.)
b) An Entry shall not exit data across the boundary, or read or write data from/to persistent storage.

The following rules help to confirm the status of a candidate Entry data movement:

RULES – Entry (E)
a) The data group of a triggering Entry may consist of only one data attribute which simply informs the software that 'an event Y has occurred'. Very often, especially in business application software, the data group of the triggering Entry has several data attributes which inform the software that 'an event Y has occurred and here is the data about that particular event'.
b) Clock-ticks that are triggering events shall always be external to the software being measured. Therefore, for example, a clock-tick event occurring every 3 seconds shall be associated with an Entry moving a data group of one data attribute. Note that it makes no difference whether the triggering event is generated periodically by hardware or by another piece of software outside of the boundary of the software being measured.
c) Unless a specific functional process is necessary, obtaining the date and/or time from the system's clock shall not be considered to cause an Entry or any other data movement.
d) If an occurrence of a specific event causes the Entry of a data group

comprising up to 'n' data attributes of a particular object of interest and the FUR allows that other occurrences of the same event can cause an Entry of a data group which has values for attributes of only a sub-set of the 'n' attributes of the object of interest, then one Entry shall be identified, moving a data group comprising all 'n' data attributes.

e) When identifying Entries in a screen that enables human functional users to input data into functional processes, analyze only screens that are filled with data. Ignore any screen that is formatted but otherwise 'blank' except for possible default values, and ignore all field and other headings that enable human users to understand the input data required.

NOTE. It may be necessary to consider field and other headings when measuring FUR for changes to Entries – see section 4.4.1.

BUSINESS EXAMPLE illustrating rule c): When a functional process adds a time stamp to a record to be made persistent or to be output, no Entry is identified for obtaining the system's clock value.

Once identified, each Entry data movement can be registered by marking the corresponding cell of the Generic Software Model matrix (Appendix A) with an 'E'.

3.5.3 Identifying Exits (X)

A candidate Exit data movement must comply with the following principles:

PRINCIPLES – Exit (X)

a) An Exit shall move a single data group describing a single object of interest from the functional process of which the Exit forms part across the boundary to a functional user. If the output of a functional process comprises more than one data group, identify one Exit for each unique data group in the output. (See also section 3.5.7 on 'Data movement uniqueness'.)

b) An Exit shall not enter data across the boundary, or read or write data from/to persistent storage.

The following rules might be useful to confirm the status of a candidate Exit data movement:

RULES – Exit (X)

a) An enquiry which outputs fixed text, (where 'fixed' means the message contains no variable data values, e.g. the result of pressing a button for 'Terms & Conditions' on a shopping web-site), shall be modeled as having one Exit for the fixed text output.

NOTE: For the output from 'Help' functionality, see the 'Guideline for sizing Business Application Software'. For the output of messages concerned with error conditions or confirming success, see section 3.5.11 of this Measurement Manual.

b) If an Exit of a functional process moves a data group comprising up to 'n' data attributes of a particular object of interest and the FUR allows that the functional process may have an occurrence of an Exit that moves a data group which has values for attributes of only a sub-set of the 'n' attributes of the object of interest, then one Exit shall be identified, moving a data group comprising all 'n' data attributes.

c) When identifying Exits, ignore all field and other headings that enable human

users to understand the output data.

NOTE: It may be necessary to consider field and other headings when measuring FUR for changes to Exits – see section 4.4.1

Once identified, each Exit data movement can be registered by marking the corresponding cell of the Generic Software Model matrix (Appendix A) with an 'X'.

See also section 3.5.11 for how to identify Exit data movements for error messages.

3.5.4 Identifying Reads (R)

A candidate Read data movement must comply with the following principles:

PRINCIPLES – Read (R)
a) A Read shall move a single data group describing a single object of interest from persistent storage to a functional process of which the Read forms part. If the functional process must retrieve more than one data group from persistent storage, identify one Read for each unique data group that is retrieved. (See also section 3.5.7 on 'Data movement uniqueness'.)
b) A Read shall not receive or exit data across the boundary or write data to persistent storage.
c) During a functional process, movement or manipulation of constants or variables which are internal to the functional process and that can be changed only by a programmer, or computation of intermediate results in a calculation, or of data stored by a functional process resulting only from the implementation, rather than from the FUR, shall not be considered as Read data movements.
d) A Read data movement always includes any 'request to Read' functionality (so a separate data movement shall never be counted for any 'request to Read' functionality). See also section 3.5.9.

The following rules might be useful to confirm the status of a candidate Read data movement:

RULES – Read (R)
a) Identify a Read when, according to the FUR, the software being measured must retrieve a data group from persistent storage.
b) Do not identify a Read when the FUR of the software being measured specify any software or hardware functional user as the source of a data group, or as the means of retrieving, a persistently-stored data group. (For this case see the principles and rules for Entries and Exits.)

Once identified, each Read data movement can be registered by marking the corresponding cell of the Generic Software Model matrix (Appendix A) with an 'R'.

3.5.5 Identifying Writes (W)

A candidate Write data movement must comply with the following principles:

PRINCIPLES – Write (W)
a) A Write shall move a single data group describing a single object of interest from the functional process of which the Write forms part to persistent

storage. If the functional process must move more than one data group to persistent storage, identify one Write for each unique data group that is moved to persistent storage. (See also section 3.5.7 on 'Data movement uniqueness'.)

b) A Write shall not receive or exit data across the boundary, or read data from persistent storage.

c) A requirement to delete a data group from persistent storage shall be measured as a single Write data movement.

d) The following shall not be considered as Write data movements:

- The movement or manipulation of any data that did not exist at the start of a functional process and that has not been made persistent when the functional process is complete;

- Creation or update of variables or intermediate results that are internal to the functional process;

- Storage of data by a functional process resulting only from the implementation, rather than from the FUR. (An example would be the storage of data temporarily during a large sort process in a batch-processed job.)

The following rules might be useful to confirm the status of a candidate Write data movement:

RULES – Write (W)
a) Identify a Write when, according to the FUR, the software being measured must move a data group to persistent storage.
b) Do not identify a Write when the FUR of the software being measured specify any software or hardware functional user as the destination of the data group or as the means of storing the data group. (For this case see the principles and rules for Entries and Exits.)

Once identified, each Write data movement can be registered by marking the corresponding cell of the Generic Software Model matrix (Appendix A) with a 'W'.

3.5.6 *On the data manipulations associated with data movements*

Sub-processes are, as defined in principle (d) of the Generic Software Model (see section 1.3), either data movements or data manipulations. However, by a COSMIC convention (see principle (j) of the Generic Software Model), the separate existence of data manipulation sub-processes is not recognized.

All data manipulation is considered to be accounted for by the associated data movements. Hence data manipulation can be ignored EXCEPT if there is a FUR that must be measured for a *change* to data manipulation. For these cases we may need the rules of this section to determine which type of data manipulation is associated with which type of data movement.

A typical Change Request affects both the data attributes moved and the data manipulation associated with a particular data movement. The rules for measuring required changes of FUR (see section 4.4 and the definition of a 'Modification' in 4.4.1) are that if any attribute(s) of the group moved by a data movement or any of its associated data manipulation must be changed, then the data movement must be identified and measured as changed. So if a required change affects only the manipulation of data, we need the following rules to identify their associated data movement(s) that must be measured as changed,

DEFINITION – Data manipulation
Anything that happens to data when it is processed by a functional process other than movement of data into or out of a functional process, or between a functional process and persistent storage.

The following principle and rules determine how the COSMIC method deals with data manipulation.

PRINCIPLE – Data manipulation associated with data movements
All data manipulation in a functional process shall be associated with the four types of data movement (E, X, R, and W). By convention, the data movements of a functional process are assumed also to account for the data manipulation of the functional process.

RULES – Data manipulation associated with data movements
a) An Entry data movement accounts for all data manipulation to enable a data group to be entered by a functional user (e.g. formatting and presentation manipulations) and to be validated.
b) An Exit data movement accounts for all data manipulation to create the data attributes of a data group to be output and/or to enable the data group to be output (e.g. formatting and presentation manipulations) and to be routed to the intended functional user.
c) A Read data movement accounts for all computation and/or logical processing needed in order to retrieve a data group from persistent storage.
d) A Write data movement accounts for all computation and/or logical processing to create or to update a data group to be written, or to delete a data group.
e) The data manipulation associated with any of these data movements does not include any data manipulation that is needed after the data movement has been successfully completed, nor does it include any data manipulation associated with any other data movement.

BUSINESS EXAMPLE 1: An Entry includes all manipulation needed to format a screen to enable a human user to enter data and to validate the entered data EXCEPT any Read(s) that might be required to validate some entered data or codes, or to obtain some associated code descriptions.

BUSINESS EXAMPLE 2: An Exit includes all manipulation to format output and prepare some data attributes for printing (or output on a screen), including the human-readable field headings[11] EXCEPT any Read(s) or Entries that might be required to supply the values or descriptions of some of the printed data attributes.

3.5.7 Data movement uniqueness and possible exceptions

Most often, in any one functional process *all* data describing any one object of interest that is required by that functional process, is input in one Entry data movement type and/or read in one Read data movement type and/or written in one Write data movement type and/or output

---

[11] This example applies when measuring application software for use by humans, regardless of the domain. It would obviously not apply when measuring the size of re-usable objects which support the display of individual field headings on input or output screens.

in one Exit data movement type. The model further assumes that all data manipulation resulting from all possible values of the data attributes of a data group that is moved is associated with the one data movement. Also, two data movements cannot be considered as different due to their having different FUR for their associated data manipulation, as 'all data manipulation in a functional process shall be associated with the four types of data movement (E, X, R and W)' (see the principle stated in section 3.5.6).

EXAMPLE illustrating this latter principle: Consider two occurrences of a given functional process (-type). Suppose that in the first occurrence the values of some attributes to be moved lead to a data manipulation sub-process (-type) 'A' and that in another occurrence of the same functional process the attribute values lead to a different data manipulation sub-process (-type) 'B'. In such circumstances, both data manipulation sub-processes 'A' and 'B' should be associated with the same one data movement and hence only the one data movement should be identified and counted in that functional process.

There can, however, be circumstances in which *different* data groups describing the *same* object of interest may be required (in the FUR) to be moved in a data movement of the same type (E, R, W, X) in the same functional process.

The following rules cover the most common situation (rule a)) and other possible valid cases (rules b) and c)). Rule d) concerns *occurrences* (as opposed to the *types*) of data movements.

RULES – Data movement uniqueness and possible exceptions
N.B. All COSMIC rules concern types of functional users, data groups, data movements, functional processes and objects of interest. For ease of reading, we normally omit 'type' from these terms. This convention is followed in rules a), b) and c) below but In rule d) we include 'type' where it is helpful to distinguish a 'type' from an 'occurrence'.

a) Unless the Functional User Requirements are as given in rules b) or c), all data describing any one object of interest that is required to be entered into one functional process shall be identified as one data group moved by one Entry.

   NOTE: A functional process may, of course, have multiple Entries, each moving data describing a different object of interest.

   The same equivalent rule applies to any Read, Write or Exit data movement in any one functional process.

b) If Functional User Requirements specify that different data groups must be entered into one functional process, each from a different functional user, where each data group describes the same object of interest then one Entry shall be identified for each of these different data groups.

   The same equivalent rule applies for Exits of data to different functional users from any one functional process.

   NOTE: Any one functional process shall have only one triggering Entry.

c) If Functional User Requirements specify that different data groups must be moved from persistent storage into one functional process, each describing the same object of interest, then one Read shall be identified for each of these different data groups.

   The same equivalent rule applies for Writes in any given functional process.

   NOTE: This rule is analogous to rule b). In the case of the FUR to read

> different data groups desrbing the same object of interest, they will likely have originated from different functional users. In the case of the FUR to write different data groups, they will likely be made available to be read by different functional users.
>
> d) Repeated occurrences of any data movement type when it is being executed shall not be counted.
>
> This applies even if multiple occurrences of the data movement type differ in their execution because different values of the data attributes of the data group moved result in different processing paths being followed through the functional process type.

The following examples illustrate the above rules.

BUSINESS EXAMPLE 1 for rule a): The most common case is that one Write would be identified that moves a data group containing <u>all</u> the data attributes of an object of interest required to be made persistent in a given functional process.

BUSINESS EXAMPLE 2 for rule c) in contrast to rule a) and Business Example 1: Suppose FUR for a single functional process A to store two data groups derived from a bank's current account files for later use by separate programs. The first data group is 'overdrawn account' details' (which includes the negative balance attribute). The second data group is 'high value account' details' (which only has the account holder's name and address, intended for a marketing mail-shot). Functional process A will have two Writes, one for each data group, both describing the same object of interest 'account'.

REAL-TIME EXAMPLE 3 for rule b): A functional process is required to accept different data groups from two different seismometers (functional users) each describing the same object of interest (the event), e.g. a test explosion. Identify two Entries.

BUSINESS EXAMPLE 4 for rule b): Suppose FUR exist for a single functional process to produce two or more Exits moving different data groups describing the same object of interest, intended for different functional users. For example, when a new employee joins a company a report is produced to be passed to the employee to sign off his personal data as valid, and a message is sent to Security to authorize the employee to enter the building. Identify two Exits.

BUSINESS EXAMPLE 5 for rule c): Suppose a FUR for a program to merge two files of persistent data describing the same object of interest, e.g. a file of existing data about an object of interest 'X' and a file with newly-defined attributes describing the same object of interest 'X'. Identify two Reads, one for each file, for the functional process.

BUSINESS EXAMPLE 6 for rule c): Suppose a Read of a data group is required in the FUR, but the developer decides to implement it by two commands to retrieve different sub-sets of data attributes of the same object of interest from persistent storage at different points in the functional process. Identify only one Read.

BUSINESS EXAMPLE 7 for rule d): See section 3.5.2, rule d) for Entries, and section 3.5.3, rule b) for Exits. (These two rules concern cases where occurrences of Entries or Exits have only a sub-set of the maximum number of data attribute-types that could be moved according to the FUR.)

BUSINESS EXAMPLE 8 for rule d): Suppose a Read is required in the FUR that in practice requires many retrieval occurrences, as in a search through a file. Identify only one Read.

REAL-TIME EXAMPLE 9 for rule d): Suppose in a real-time functional process, the FUR requires that the same identical data group must be entered from a given functional user, e.g. a hardware device, twice at a fixed time interval in order to measure a rate of change

during the process. The two movements of data are therefore multiple occurrences of the same Entry. Only one Entry may be identified for this data group for this functional process. (There is no data manipulation associated with the two occurrences of the Entry. The calculation of the rate of change must be associated with the Exit that reports the rate. Refer to section 3.5.6 for the types of data manipulation that are considered to be associated with an Entry.)

REAL-TIME EXAMPLE 10 for rule d): Suppose a process control system for a machine that produces a flat product such as paper or a plastic film. The machine has an array of 100 identical sensors across the direction of movement of the product to detect breaks or holes in the product. The functional process that must check for breaks or holes receives the same data from each sensor. The position of e.g. a hole in the product can be determined e.g. from the position in the string of data values sent from the array of sensors, or by each sensor sending its ID, etc. Either way, the processing of data from all the sensors is identical. Identify only one functional user type for all the sensors and only one Entry type for the data obtained from all the sensors by the functional process.

3.5.8 When a functional process is required to move data to or from persistent storage

When the FUR require data to be stored or to be retrieved from storage, the Measurer must investigate whether the data can be stored or retrieved within its own boundary, i.e. to/from 'persistent storage', or whether data is required to be stored/retrieved with help of a functional user of the software being measured (i.e. via some other piece of software, or directly to or from a hardware device). This section describes the data movements involved when a functional process of a piece of software is required to obtain some data from persistent storage, via four examples below:

- Example 1 is typical of application software but could apply to software in any layer, except for the layer where software interacts directly with a physical hardware store. The functional process is required to retrieve data from persistent storage but the FUR are not concerned with how those data accesses are handled by any other software;

- Example 2 is for software with a 'client-server' architecture where a functional process of a client must request the server for some persistent data;

- Example 3 applies where different pieces of software have different access rights to persistent data;

- Example 4 applies where the data must be obtained directly from a physical hardware store perhaps by device driver software.

The examples are illustrated using the conventions of Message Sequence Diagrams. The notation of these diagrams is as follows:

- A bold vertical arrow pointing downwards represents a functional process.

- Horizontal arrows represent data movements, labelled E, X, R or W for Entry, Exit, Read and Write, respectively. Entries and Reads are shown as arrows incoming to the functional process and Exits and Writes as outgoing arrows; as far as possible they appear in the sequence required, from top to bottom, of the functional process.

- A vertical dotted line represents a boundary.

EXAMPLE 1: When a functional process is required to move a data group to or from persistent storage. This example concerns a piece of software A that is required to retrieve a stored data group where the FUR of software 'A' are not concerned with how those data accesses are handled by any other software in the same or different layers.

The functional users of the software A could be, for example, human users if the software A is in the application layer making an enquiry on a stored data group. Figure 3.6 shows

the COSMIC data movements for this enquiry. The enquiry is triggered by an Entry, followed by a Read of the data group from persistent storage and then an Exit with the enquiry result. FP A is not concerned with where the data is retrieved from, only that it is persistent data.
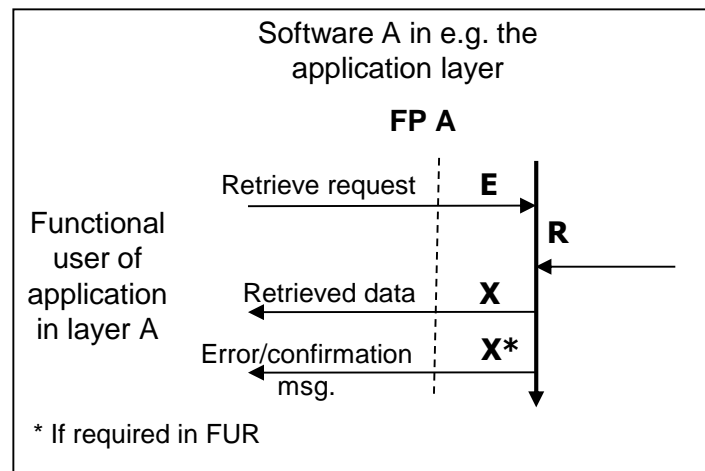
Figure 3.6 – Solution for a Read issued by software 'A' in the application layer

An exactly analogous model would apply if the functional process FP A were required to make a data group persistent via a Write data movement. According to rule d) for error conditions (see section 3.5.11), the Read and the Write data movements are considered to account for any return code or reporting of an error condition.

Figure 3.6 shows a possible application-specific error message which could be issued by FP A if, for example, the requested record is not found. However, an error condition not specific to the application, e.g. 'disk failure' would not be counted as an Exit for FP A. See also section 3.5.11 on error/confirmation messages.

EXAMPLE 2: When a functional process is required to obtain some data from another piece of software.

In this example, the pieces of software to be measured are assumed to have a 'client/server' relationship, i.e. where one piece, the client, obtains services and/or data from the other piece, the 'server', in the same or a different layer. Figure 3.7 shows an example of such a relationship, in which the two pieces are major components of the same application. In any such client/server relationship, the FUR of the client component C1 would identify the server component C2 as one of its functional users, and vice versa. The same relationship would exist and the same diagram would apply if the two pieces were separate applications, or if one of the pieces were a component of a separate application.

Physically, the two components could execute on separate processors; in such a case they would exchange data via the respective operating systems and any other intervening layers of their processors in a software architecture such as shown in Figure 2.2. But logically, applying the COSMIC models, the two components exchange data via an Exit followed by an Entry data movement. All intervening software and hardware is ignored in this model. (See also the right-hand side of Figure 3.1 for a similar example.).

Figure 3.7 shows that a functional process FP C1 of the client component C1 is triggered by an Entry from a functional user (such as a human) which consists, for example, of the parameters of the enquiry. The FUR of component C1 will recognize that this component

must ask the server component C2 for the required data, and must tell it what data group is required.

To obtain the required data group, FP C1 issues an Exit containing the enquiry request parameters to component C2. This Exit data movement crosses the boundary between C1 and C2 and so becomes the triggering Entry of a functional process FP C2 in the component C2. The functional process FP C2 of component C2 is assumed to obtain the required data group via a Read from its own persistent storage, and sends the data back to C1 via an Exit. Functional process FP C1 of component C1 receives this data as an Entry. FP C1 then passes the data group on as an Exit to satisfy the enquiry of its functional user.

Taking into account the possible error/confirmation message issued by the client, this Example 2 enquiry therefore requires 5 data movements (i.e. 5 CFP) to satisfy the enquiry request for component C1 and 3 CFP for component C2. This compares with the 4 CFP (1 x E, 1 x R and 2 x X) that would have been required for component C1 if it had been able to retrieve the data group from persistent storage within its own boundary via a Read as shown in Figure 3.7.
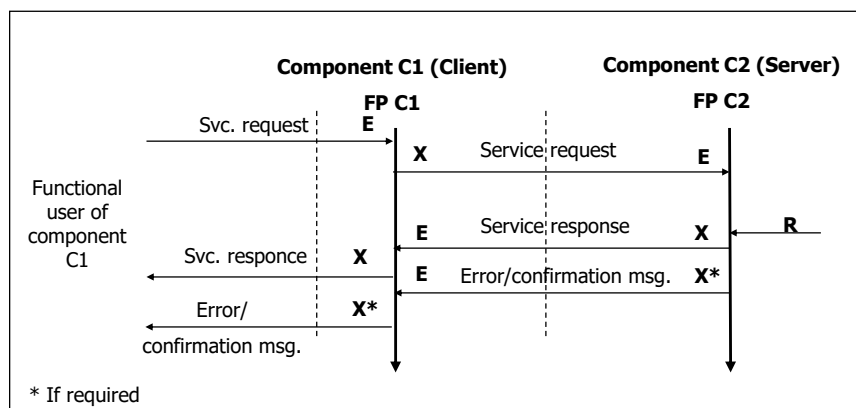
Figure 3.7 – Data exchanges between client and server components

Component C2 will probably, of course, use the services of some storage device driver software in another layer of the software architecture to retrieve the data from the hardware, as in Example 4, Figure 3.9 (b).

Examples 1 and 2 illustrate the data movements when it is clear from the FUR that the software being measured must access the persistent storage within its own boundary, or must pass the access request to another piece of software outside its boundary, respectively. Sometimes, however, the piece of software being measured may have to use different 'routes' to access the persistent data depending on the specific data attributes to be accessed and/or the type of access (storage or retrieval). This can arise when access to data by the software being measured is subject to varying rules or 'rights' due to issues of e.g. security or privacy (see Example 3 below), or the need to ensure data integrity by restricting access for all create, update and delete processes. Where the FUR of the software being measured are unclear on this point, the Measurer must take care to determine the actual rights of access. (Do NOT confuse 'right of access' to data with data 'ownership'; the latter is irrelevant to the COSMIC model. Persistent storage is not 'owned' by any piece of software.)

Note: the example of Figure 3.7 assumes synchronous communication between the client and the server, i.e. the client waits for the response from the server. Alternatively, the communication could take place asynchronously. This case would be modelled slightly differently but the number of data movements would be the same as in Figure 3.7 (see the 'Guideline for sizing Business Application software' [7]).

Note: in practice the client-server communication may be subject to timeout monitoring in case the server does not respond within a required time. For an example of how this may be modelled and measured, see the 'Guideline for sizing Real-Time software' [4].

EXAMPLE 3: The software has different rights of access to stored data for different purposes.

See Figure 3.8. A piece of software A to be measured is allowed to retrieve certain stored data Z (as in Example 1, Figure 3.6), but it is not allowed to maintain (i.e. create, update or delete) this same data Z directly. When the software A is required to maintain the data Z, software A must pass its request to another piece of software B via an Exit followed by an Entry (analogous to Example 2, Figure 3.7 of component C1 passing its request to component C2). The piece of software B is required to ensure the integrity of the data Z by ensuring consistent validation, so it processes all data maintenance requests for the data Z.
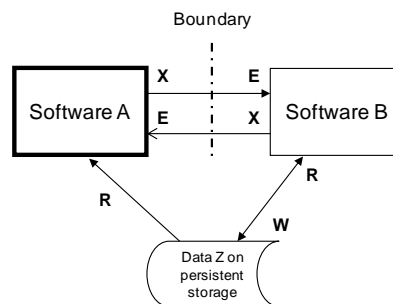
Figure 3.8 – Persistent data Z within the boundary of both software A and B for a Read

In this Example 3, COSMIC models would show that the data Z is held on persistent storage within the boundary of the piece of software A but only for retrieval purposes where it can be accessed by Read data movements. For software B, this same data Z is held on persistent storage within its boundary and software B can both Read and Write this data Z. For error handling in this Example, see Examples 1 and 2 above.

INFRASTRUCTURE EXAMPLE 4: How persistent data is obtained by the device driver software that interacts with the physical storage device.

This example concerns the piece of software A of Example 1 that is required to retrieve a stored data group. We also consider a separate piece of software 'B' that is the device driver for the intelligent hardware storage which holds the data group that the piece of software A is required to access. (We ignore the probable presence of an operating system for simplicity; the operating system effectively transmits application requests to the device driver software and returns the results of requests.)

The two pieces of software are in different layers in an architecture such as shown in Figure 2.2. Software A is in e.g. the application layer, and software B is in a device driver layer. Physically, there is probably a hierarchical relationship between the two pieces and (ignoring the operating system) a physical interface between software in the two layers, as shown for example in Figure 2.2. However, the models of the functional processes of software A and B are independent of the nature of the relationship between the layers, which may be hierarchical or bi-directional.

The functional users of the software B in the driver layer are the piece of software A (ignoring the operating system) and the intelligent hardware storage device which holds the required data. ('Intelligent' means that the device must be told what data is needed.)

Suppose that an enquiry functional process FP A of the software A needs to retrieve a stored data group. Figure 3.9 (a) shows the COSMIC model of this enquiry. Figure 3.9 (b)

shows the functional process FP B of the software B in the device driver layer that handles the physical retrieval of the required data from a hardware storage device (such as a disk or USB memory stick).
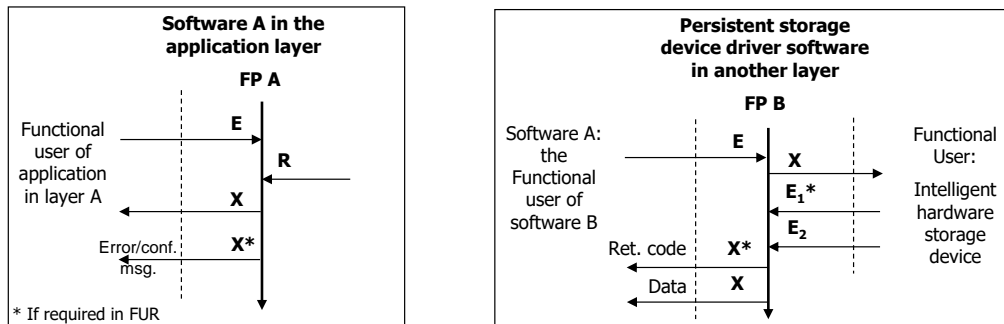
Figures 3.9 (a) and (b) – Solution for a Read issued by software A in the application layer to software B in the device driver layer

Figure 3.9 (b) shows that the Read request of the software A is received as a triggering Entry to the functional process FP B, which passes on the request as an Exit to the hardware device. The response of the latter depends on the particular hardware device. The device may just return the requested data, shown as Entry E_2 in Figure 3.9 b). The device may also issue a separate error message describing the success or the reason for the failure of the request, e.g. 'data not found', or disk error', shown as Entry E_1* in Figure 3.9 b). FP B returns the data to the software A as an Exit. FP B also normally issues a 'return code' describing the success or reason for the failure of the request. (Although the return code may be physically attached to the returned data, it is logically a different data group to that of the returned data – it is data about the outcome of the request process). For FP A no Entry for these messages is identified, as the Read data movement accounts for the retuned data and error messages, according to rule d) for an Entry. For FP A, an Exit is identified for an error/confirmation message, if required.

Note: in practice, there may be more data movements between the device driver software and the intelligent hardware device than are shown in Fig 3.8 b). For example, this Figure does not show the effect of the device driver measuring a timeout for non-response from the hardware.

Comparing Examples 2 and 4, we see that in Example 4 the models of the piece of software A and the device driver B of Example 4 cannot be combined as they are in Example 2. This is because A and B are in different layers and a Read does not cross a boundary. Figure 3.9 (b) shows that the software A is a functional user of the device driver software B. But the reverse is not true, because a Read does not cross a boundary. In contrast, Figure 3.7 can show the two components in one model because Component C1 is a functional user of component C2, and vice versa, and they share a common boundary.

3.5.9 *When a functional process requires data from a functional user*

If a functional process must obtain data from a functional user there are two cases. If the functional process does not need to tell the functional user what data to send, a single Entry is sufficient (per object of interest). If the functional process needs to tell the functional user what data to send, an Exit followed by an Entry are necessary. The following rules apply:

RULES – A functional process requiring data from a functional user
a) A functional process shall obtain a data group via an Entry data movement from a functional user, when the functional process does not need to tell the

functional user what data to send, as in any of the following four cases:

- when a functional user sends a data group via a triggering Entry which initiates the functional process;

- when a functional process, having received a data group via a triggering Entry, waits, expecting the arrival of a further data group from the functional user via an Entry (can occur when a human functional user enters data to business application software);

- when a functional process, having started, requests the functional user, 'send me your data now, if you have any' and the functional user sends its data;

- when a functional process, having started, inspects the state of a functional user and retrieves the data it requires.

In the latter two cases (typically occurring in real-time 'polling' software), by convention no Exit from the functional process shall be identified to obtain the required data. The functional process merely needs to send a prompt message to a functional user and the functionality of that prompt message is considered to be part of the Entry. The functional process knows what data to expect. Only one Entry shall be identified for this case.

b) Where a functional process needs to obtain the services of a functional user (for instance to obtain data) and *the functional user needs to be told what to send* (typically where the functional user is another piece of software outside the scope of the software being measured), an Exit followed by an Entry data movement shall be identified. The Exit sends the request for the specific data; the Entry receives the returned data.

REAL-TIME EXAMPLE 1 of rule a), third or fourth bullet: Suppose a functional process of a real-time process control application software system is required to poll an array of identical dumb sensors. At the application level, the request for the data by the functional process and the receipt of the data is accounted for by one Entry (type). (Since the sensors are identical only one Entry (type) is identified and counted although there are multiple occurrences.)

Suppose further that the request for the data must in practice be passed to a piece of device driver software in a lower layer of the software architecture, which physically obtains the required data from the sensor array as illustrated in the layered architecture of Figure 2.3. The functional processes of the process control application software and of the device driver software for the dumb sensors would be as shown in Figures 3.10 (a) and (b) below.
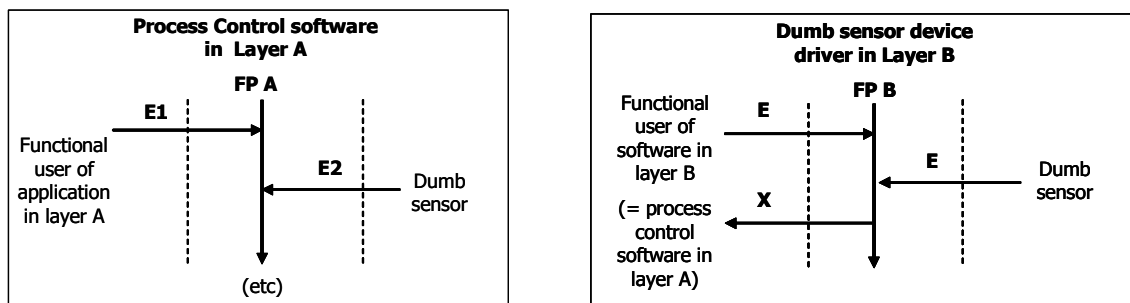
Figure 3.10 (a) and (b) – Solution for a poll of dumb sensors issued by software A in the process control application layer handled by software B in the dumb sensor device driver layer

Figure 3.10 (a) shows that the application software functional process FP A is triggered by an Entry E1 e.g. from a clock tick. This functional process then obtains data via Entry E2 from the dumb sensor array to receive the multiple occurrences of the sensor readings. The dumb sensors are also functional users of the process control software in this application level model. (The device driver software is hidden at this level.)

Figure 3.10 (b) shows the model for the software that drives the dumb sensor devices. It receives data via an Entry from the application (probably in practice via an operating system) as the trigger of a functional process FP B. This functional process obtains the required data via an Entry E from its functional user, the dumb sensor array.

The data group is passed back to the process control software via an Exit. This Exit is received as the Entry E2 by the application functional process FP A. FP A then continues with its processing of the sensor data. Again, the fact that there are multiple occurrences of this cycle of gathering data from each of the identical sensors is irrelevant to the model.

The apparent mis-match between the one Entry E2 from a dumb sensor to the process control application software and the Entry followed by an Exit data movement of the device driver software is due to the convention that an Entry from a dumb sensor is considered to include any 'request to enter' functionality since the dumb functional user has no capability of dealing with any message from a functional process.

REAL-TIME EXAMPLE 2 of rule b): Suppose a functional process sends to one of its functional users, such as an 'intelligent' hardware device or another peer piece of software, some parameters for an enquiry or the parameters for a calculation, or some data to be compressed. The response from the functional user is obtained via the functional process issuing an Exit, followed by the receipt of an Entry data movement, as described in section 3.5.8, Example 2.

3.5.10 Navigation and display control commands for human users ('control commands')

A 'control command' is a command that is recognized in any application which may be used by human functional users and which must be ignored when measuring a functional size. The definition is:

DEFINITION – Control command
A command that enables human functional users to control their use of the software but which does not involve any movement of data about an object of interest of the FUR of the software being measured.
NOTE: A control command is not a data movement because the command does not move data about an object of interest.

RULE – Control commands in applications with a human interface
In an application with a human interface 'control commands' shall be ignored as they do not involve any movement of data about an object of interest.

EXAMPLES OF CONTROL COMMANDS

- *Commands to 'page up/down' or between physical screens.*

- *Hitting a Tab or Enter key, or pressing a button to continue.*

- *Clicking on an 'OK' button to confirm or cancel a previous action, or to acknowledge an error message or to confirm some entered data, etc.*

- *Functions that enable a user to control the display (or not) of a header or of sub-totals that have been calculated;*
- *Menu commands that enable a user to navigate to one or more specific functional processes but which do not themselves initiate any one functional process,*
- *Commands to display a blank screen for data entry.*

N.B. Outside the domain of applications with a human user interface, the concept of a 'control command' has no special meaning and any signal or movement of data about an object of interest coming from a functional user must be accounted for, i.e. must be measured.

3.5.11 Error/Confirmation Messages and other indications of error conditions

DEFINITION – Error/confirmation message
An Exit issued by a functional process to a <u>human</u> functional user that either confirms only that entered data has been accepted, or only that there is an error in the entered data.
NOTE: Any Exit that may include fault indications, but that is not intended for a human functional user, is not an error/confirmation message.

RULES – Error/confirmation messages and other indications of error conditions
a) One Exit shall be identified to account for all types of error/confirmation messages issued by any one functional process of the software being measured from all possible causes according to its FUR, e.g. successes or failures of: validation of entered data or for a call to retrieve data or to make data persistent, or for the response from a service requested of another piece of software. NOTE: If the FUR of the functional process do not require any type of error/confirmation message to be issued, do not identify any corresponding Exit.
b) If a message to a <u>human</u> functional user provides data in addition to confirming that entered data has been accepted, or that entered data is in error, then this additional data should be identified as a data group moved by an Exit in the normal way, in addition to the error/confirmation Exit.
c) All other data, issued or received by the software being measured, to/from its <u>hardware or software</u> functional users should be analyzed according to the FUR as Exits or Entries respectively, according to the normal COSMIC rules, regardless of whether or not the data values indicate an error condition.
d) Reads and Writes are considered to account for any associated reporting of error conditions. Therefore no Entry to the functional process being measured shall be identified for any error indication received as a result of a Read or Write of persistent data.
e) No Entry or Exit shall be identified for any message indicating an error condition that might be issued whilst using the software being measured but which is not required to be processed in any way by the FUR of that software, e.g. an error message issued by the operating system.

BUSINESS EXAMPLE 1 illustrating rule a): In a human-computer dialogue, examples of error messages occurring during validation of data being entered could be 'format error', 'customer not found', 'error: please tick check box indicating you have read our terms and conditions', 'credit limit exceeded', etc. All such error messages should be considered as occurrences of one Exit in each functional process where such messages occur (which could be named 'error messages').

BUSINESS EXAMPLE 2 illustrating rule a): functional process 'A' can potentially issue 2 distinct confirmation messages and 5 error messages to its functional users. Identify one Exit to account for all these (5 + 2 = 7) error/confirmation messages. Functional process 'B' can potentially issue 8 error messages to its functional users. Identify one Exit to account for these 8 error messages.

BUSINESS EXAMPLE 3 illustrating rules a) and b): A functional process of a bank's ATM (i.e. an 'automatic teller machine', or 'cash dispenser') can issue five types of messages in response to a request to withdraw a specific amount of cash:

- *Error: machine has no available cash*

- *Error: the amount requested must be a multiple of $10*

- *Withdrawal refused. Account blocked. Contact the bank.*

- *Withdrawal refused (credit limit would be exceeded by $139.14)*

- *Withdrawal accepted; your remaining balance is $756.25*

The first four messages describe an error condition and the first part of the fifth message is a confirmation. For all of this output, count one Exit, as per rule a). The last two messages also include data attributes related to the customer's account. Count one Exit for this data, as per rule b) above, and taking into account rule a) of the 'Data movement uniqueness rules' of section 3.5.7. In total identify two Exits for this functional process, i.e. 2 CFP for its output.

BUSINESS EXAMPLE 4 illustrating rule e): Error messages output to the human users but not generated or processed by the application software being measured should be completely ignored in the measurement of the application. An example of such a message passed on from the operating system could be 'printer X is not responding'.

REAL-TIME EXAMPLE 1 illustrating rule c): In a real-time system, a functional process that periodically checks the correct functioning of all hardware devices might issue a message that reports 'Sensor X has failed', where 'X' is a variable. This message should be identified as one Exit in that functional process.

REAL-TIME EXAMPLE 2 illustrating rule c). The FUR of REAL-TIME EXAMPLE 1 in section 3.5.9 may also state that the FP's A and B must handle an error condition when the device driver software fails to obtain the data from one or more of the array of dumb sensors. A dumb sensor cannot, by definition, issue an error message. The device driver FP B will, most likely, obtain a string of values from the array of dumb sensors, e.g. state 1, state 2, state 3, no response, state 5, no response, state 7, etc. and will issue this string as an Exit to the FP A of the application where it is received as an Entry. No separate error message should be identified as an Exit from FP B of the device driver software, nor as an Entry to FP A of the process control application.

THE MEASUREMENT PHASE

4.0 Chapter summary

This chapter discusses the final step of the measurement process. First, the COSMIC unit of measure is defined (namely one data movement is measured as 1 (one) COSMIC Function Point, or 'CFP'). Then the rules are given for assigning a size to the FUR of the software being measured. Rules are defined for how to aggregate sizes of different pieces of software.

Also, rules are defined for how to size required changes to software (as dealt with in 'enhancement' projects). Finally, the chapter discusses the possibility of 'local extensions' to the standard COSMIC method which may be used, for example, in the local environment of an organization that desires to account for some aspect of functionality in a way which is meaningful as a local standard.

4.1 The measurement phase process

The general method for measuring a piece of software when its Functional User Requirements have been expressed in terms of the COSMIC Generic Software Model is summarized in Figure 4.0 below.

COSMIC MEASUREMENT PHASE

FUR in the form of the COSMIC Generic Software Model

Section 4.2
APPLY THE COSMIC UNIT OF MEASUREMENT

Section 4.4
SIZING CHANGES

Section 4.5
LOCAL EXTENSIONS

Section 4.3
AGGREGATE MEASUREMENT RESULTS

recorded information

Functional Size of the measured software
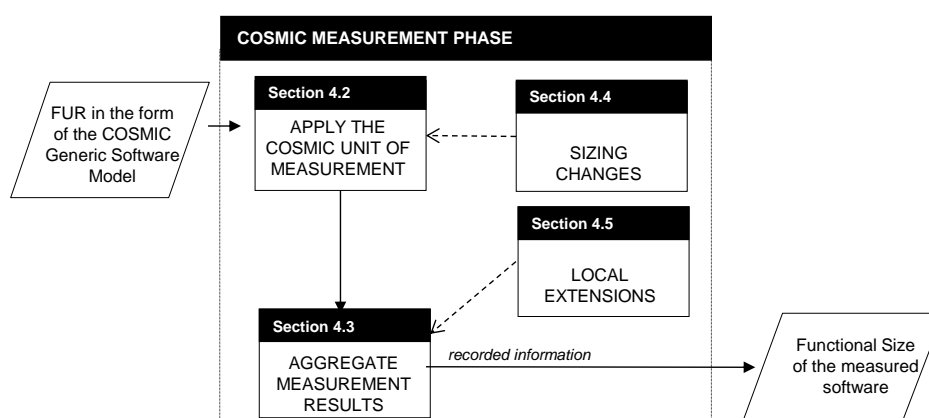
Figure 4.0 – General process for the COSMIC Measurement Phase

Each step in this method is the subject of a specific section of this chapter where the definitions and principles to apply are presented, along with some rules and examples.

4.2 Applying the COSMIC unit of measurement

DEFINITION – COSMIC unit of measurement

1 CFP (Cosmic Function Point) which is defined as the size of one data movement.

NOTE: The unit of measurement was known as a 'Cfsu' (COSMIC functional size unit.), prior to v3.0 of the method

Arising from this definition, each data movement (Entry, Exit, Read or Write) that is required to be added, modified or deleted for the software being measured is also measured as 1 CFP.

4.3 Aggregating measurement results

This step consists in aggregating the sizes of all identified data movements, into a single functional size value. This step is accomplished according to the following rules.

4.3.1 General rules of aggregation

RULES – Aggregating measurement results

a) For any functional process, the functional sizes of individual data movements shall be aggregated into a single functional size value in units of CFP by adding them together.

$$\text{Size (functional process)} = \Sigma \text{ size(Entries)} + \Sigma \text{ size(Exits)} + \Sigma \text{ size(Reads)} + \Sigma \text{ size(Writes)}$$

b) For any functional process, the functional size of changes to its Functional User Requirements shall be aggregated from the sizes of the data movements that have been added, modified or deleted in the functional process to give a size of the change in units of CFP, according to the following formula.

$$\text{Size (Change(functional process))} = \Sigma \text{ size (added data movements)} + \Sigma \text{ size (modified data movements)} + \Sigma \text{ size (deleted data movements)}$$

For more on aggregating functional size, see section 4.3.2. For measuring the size of the changed software, see section 4.4.2.

c) The size of a piece of software within a defined scope shall be obtained by aggregating the sizes of the functional processes for the piece, subject to rules e) and f) below.

d) The size of any change to a piece of software within a defined scope shall be obtained by aggregating the sizes of all changes to all functional processes for the piece, subject to rules e) and f) below

e) Sizes of pieces of software or of changes to pieces of software may be added together only if measured at the same functional process level of granularity of their FUR.

f) Sizes of pieces of software and/or changes in the sizes of pieces of software within any one layer or from different layers shall be added together only if it makes sense to do so, for the purpose of the measurement.

g) The size of a piece of software is obtained by adding up the sizes of its components (regardless of how the piece is decomposed) and eliminating the size contributions of inter-component data movements.

h) Only one Exit shall be identified for all error/confirmation messages issued by any one functional process to a human functional user.

i) If the COSMIC method is extended locally (for example to measure some aspect of size not covered by the standard method), then the size measured via the local extension shall be reported separately as described in section

---

> 5.1 and shall NOT be added to the size obtained by the standard method, measured in CFP (see further in section 4.5).

EXAMPLE 1 for rules b) and c): A required change to a piece of software might be: 'add one new functional process of size 6 CFP, and in another functional process add one data movement, make modifications to three other data movements and delete two data movements.' The total size of the required change is 6 + 1 + 3 + 2 = 12 CFP.

EXAMPLE 2 for rule f): If various major parts of a piece of software are developed using different technologies, by different project sub-teams, there may be no practical value in adding their sizes together.

EXAMPLE 3 for rule g): If a piece of software is

- *first measured 'as a whole', i.e. all within one scope*
- *then secondly the size of each of its components is measured separately, i.e. each with its own scope,*

then the total size from adding up the size of all the separate components (in the second case) will exceed the size when measured 'as a whole' (in the first case) due to the contribution to size of all the inter-component data movements. These inter-component data movements are not visible when the piece is measured 'as a whole'. See also the example in the section on measuring at varying levels of granularity in pure software architectures in the 'Guideline for Early or Rapid COSMIC Functional Size Measurement' [6])

It is to be noted that, *within* each identified layer, the aggregation function is fully scalable. Therefore a sub-total can be generated for individual functional processes or for all the software within a layer, depending on the purpose and scope of each measurement exercise and subject to rules d), e) and f) above.

4.3.2 More about functional size aggregation

In a context where functional size is to be used as a variable in a model, for example to estimate effort, and the software to be measured extends over more than one layer, aggregation will typically be performed per layer since software in different layers is often not implemented with the same technology.

EXAMPLE 1: Consider software where the application layer is to be implemented using a third generation language and a set of existing libraries, while a driver layer might be implemented using assembly language. The effort-per-unit-size associated with the construction of the software in each layer will, most probably, be different, and, consequently, an effort estimate will be prepared separately for the software in each layer. It is unlikely there will be any value in adding the sizes of the software in the two layers

EXAMPLE 2: If a project team has to develop a number of major pieces of software and is interested in its overall productivity, it can add together the work-hours needed to develop each piece. Similarly, the team can add together the sizes of the major pieces it has developed if (but only if) those sizes satisfy the rules given above.

The reason that sizes of pieces of software from different layers of a standard layered architecture, measured at the same functional process level of granularity, may be added together if it makes sense to do so (e.g. all pieces of software are developed using the same technology) is that such an architecture has a coherently defined set of functional users. Software in each layer is a functional user of software in the other layers that it uses and any piece of software in a layer may be a functional user of any of its peer pieces of software. It is therefore only logical that the sizes of the various pieces may be added together, provided always that rules d), e) and f) above apply. However, in contrast, the size of any piece of

software may *not* be obtained by adding up the sizes of its component re-usable objects unless the inter-object data movements are eliminated, as per rule g) above.

Aggregating the measurement results by type of data movement might be useful for analyzing the contribution of each type to the total size of the software in a given a layer, and might thus help characterize the functional nature of the measured software in the given layer.

4.4   More on measurement of the size of changes to software

A 'functional change' to existing software is interpreted in the COSMIC method as 'any combination of additions of new data movements or of modifications or deletions of existing data movements, including to the associated data manipulation'. The terms 'enhancement' and 'maintenance'[12] are often used for what we here call a 'functional change'.

The need for a change to software may arise from either

- a new FUR (i.e. only additions to the existing functionality), or

- from a change to the FUR (perhaps involving additions, modifications and deletions) or

- from a 'maintenance' need to correct a defect

The rules for sizing any of these changes are the same but the Measurer is alerted to distinguish the various circumstances when making performance measurements and estimates.

When a piece of software is completely replaced, for instance by re-writing it, with or without extending and/or omitting functionality, the functional size of this change is the size of the replacement software, measured according to the normal rules for sizing new software. This case will not be considered further in this section. The Measurer should be aware, however, of the need when making performance measurements or estimates to distinguish between projects to develop entirely new software and projects to 're-develop' or 'replace' existing software.

Often, an obsolete part of an application is deleted ('disconnected' would be a better description) by leaving the program code in place and by just removing the link to the obsolete functionality. When the functionality of the obsolete part amounts to 100 CFP but the part can be disconnected by changing, say, 2 data movements, 100 and not 2 data movements shall be identified as the size of the functional change. We measure the size of the requirement, not the size that was implemented.

Note that for estimation purposes it may be advisable to use a different productivity for this part of the functional change, since disconnecting is quite different from 'real' deletes. Alternatively, for estimating purposes it may be preferable to measure the size that will be implemented (2 CFP in the example) rather than the size of the requirement (100 CFP in the example). If the 'project size' of 2 CFP is measured, this should be clearly documented and distinguished from a measurement of the FUR which require that the application should be reduced in size by 100 CFP.

Note the difference between the size of the functional change (discussed here) and the change in the functional size of the software. Usually, they are different. In section 4.4.2 the size of the latter is addressed.

---

[12] A normal measurement convention is that the functional size of a piece of software does not change if the software must be changed to correct a defect so as to bring the software in line with its FUR. The functional size of the software does change if the change is to correct a defect in the FUR.

4.4.1 Modifying functionality

Any data movement of a given type (E, X, R and W) involves two types of functionality: it moves a single data group and it has some associated data manipulation (for the latter, see section 3.5.6). Hence for measurement purposes a data movement is considered to be functionally modified as follows.

DEFINITION – Modification (of the functionality of a data movement)
a) A data movement is considered to be functionally modified if at least one of the following applies: • the data group moved is modified,   • the associated data manipulation is modified.   b) A data group is modified if at least one of the following applies: • one or more new attributes are added to the data group,   • one or more existing attributes are removed from the data group,   • one or more existing attributes are modified, e.g. in meaning or format (but not in their values)   c) A data manipulation is modified if it is functionally changed in any way.

EXAMPLE: A data manipulation is modified for instance by changing the calculation, the specific formatting, presentation, and/or validation of the data. 'Presentation' can mean, for example the font, background colour, field length, field heading, number of decimal places, etc.

Control commands and application-general data of business applications do not involve data movements, as no data about objects of interest are moved. Therefore, changes to control commands and application-general data should not be measured. As an example, when the screen colour for all screens is changed, this change should not be measured. (See section 3.5.10 for an explanation of control commands and application-general data.)

RULES – Modifying a data movement
a) If a data movement must be modified due to a change of the data manipulation associated with the data movement and/or due to a change in the number or type of the attributes in the data group moved, one changed CFP shall be measured, regardless of the actual number of modifications in the one data movement.
b) If a data group must be modified, data movements moving the modified data group whose functionality is not affected by the modification to the data group shall not be identified as changed data movements.
NOTE: A modification to any data appearing on input or output screens that are not related to an object of interest to a functional user shall not be identified as a changed CFP. (See section 3.3.3 for examples of such data.)

EXAMPLE for rule a): A change request for a functional process requires three changes to the data manipulation associated with its triggering Entry and two changes to the manipulation associated with an Exit, as well as two changes to the attributes of the data group moved by this Exit. Measure the size of the change as 2 CFP, i.e. count the total number of data movements whose attributes and associated data manipulation must be changed. Do NOT count the number of data manipulations or data attributes to be changed.

EXAMPLE for rules a) and b): Suppose a requirement to add or to modify the data attributes of a data group D_1, such that after modification it becomes D_2. In the functional process A where this modification is required, all data movements affected by the modification should be identified and counted as modified. So, as per rule a), if the changed data group D_2 is made persistent and/or is output in functional process A, identify one Write and/or one Exit data movement respectively as modified. However, it is possible that other functional processes Read or Enter this same data group D_2, but their functionality is unaffected by the modification because they do not process the changed or added data attributes. These functional processes continue to process the data group moved as if it were still D_1. So, as per rule (b), these data movements in the other functional processes that are not affected by the modification to the data movement(s) of functional process A must NOT be identified and counted as modified.

BUSINESS EXAMPLE: If an error/confirmation message is required to be changed (i.e. texts added, modified or deleted) it should be identified for measurement, regardless of whether or not the changed text is a consequence of a requirement to change another data movement.

4.4.2 Size of functionally-changed software

RULES – Size of functionally-changed software
After functionally changing a piece of software:

New total size (changed piece of software) = Old total size (piece of software)
 	
          +   Σ size (added data movements)
 	
          -    Σ size (deleted data movements)

Modified data movements have no influence on the size of the piece of software as they exist both before and after the modifications have been made.

EXAMPLE: Recall the Example 1 in section 4.3.1: "A required change to a piece of software might be: 'add one new functional process of size 6 CFP, and in another functional process add one data movement, make modifications to three other data movements and delete two data movements.' The total size of the required change is 6 + 1 + 3 + 2 = 12 CFP."

The total size of the piece of software will have increased by 6 CFP due to the addition of the one new functional process and will have <u>decreased</u> by 1 CFP due to the additions to (+1 CFP) and deletions from (-2 CFP) the other functional process. After the change, the piece of software will therefore have increased in size by (+6 - 1) = 5 CFP.

4.5 Extending the COSMIC measurement method

4.5.1 Introduction

The COSMIC method of measuring a functional size does not presume to measure all possible aspects of software 'size'. Thus the method is currently not designed to measure separately and explicitly the size of the FUR of data manipulation sub-processes. The influence on size of data manipulation sub-processes is taken into account via a simplifying assumption that is valid for a wide range of software domains, as defined in section 1.1 on the applicability of the method. Also, the influence of the number of data attributes per data movement on software size is not captured.

Other parameters such as 'complexity' (however defined) might be considered to contribute to functional size. A constructive debate on this matter would first require commonly-agreed

definitions of the other elements within the ill-defined notion of 'size' as it applies to software. Such definitions are still, at this point, the subject of further research and of much debate.

Nevertheless, the COSMIC size measure is considered to be a good approximation for the method's stated purpose and domains of applicability. Yet, it may be that within the local environment of an organization using the COSMIC measurement method, it is desired to account for such functionality in a way which is meaningful as a local standard. For this reason, the COSMIC measurement method has provision for local extensions. When such local extensions are used, the measurement results must be reported according to the special convention presented in section 5.1. The following sections show how to extend the method with a local standard.

4.5.2   Data manipulation-rich software

The COSMIC method was designed to measure 'data movement-rich' software. Like all other true Functional Size Measurement (FSM) methods, it was not designed to measure explicitly the functionality of data manipulation. Instead, the method assumes that the data movement types account for the associated data manipulation functionality (see further below). This assumption has proven to be reasonable for all the practical purposes such as project performance measurement and estimating for which the method was designed and for the domains in which it is commonly used.

Experience has shown however that the method can also often be successfully applied to size 'data manipulation-rich' software, e.g. some scientific/engineering software. This is true, for example, where the software must handle high volumes of data, leading to very large numbers of data movement types. The latter may effectively account for any mathematically-complex data manipulation that may also be present. By 'successfully applied', we mean that the method has produced meaningful and useful sizes in relation to the purpose of the measurement. Examples include the sizing of expert systems, software to digitally process continuous variables, software that collects and analyzes data from scientific experiments or from engineering measurements, etc.

However, given the fundamental design of the COSMIC method, users of the method, when faced with measuring a functional size of software that is rich in data manipulation, should decide for themselves whether the method really produces functional sizes that are meaningful and useful in relation to the purpose of the measurement. Where the method cannot account adequately for data manipulation, it may be possible to develop a local extension to the method to overcome the limitation – see section 4.5.5.

4.5.3   Limitations on the factors contributing to functional size

Within its domains of applicability, the COSMIC method does not attempt to measure all possible aspects of functionality that might be considered to contribute to software 'size'. For example, the measurement method does not explicitly capture the influence of software 'complexity'. But there are many types of complexity e.g. architectural, semantic, timing, process, data, etc., and in measuring functional size, the method actually accounts in a simple way for the contribution to size of process complexity (and thus indirectly of data complexity).

The method also does not consider the influence of the number of data attributes per data movement on software functional size. As described in section 4.5.6, if desired, such aspects of functional size may be supported by a local extension to the COSMIC measurement method.

4.5.4   Limitations on measuring very small pieces of software

All functional size measurement methods are based on the assumptions of a simplified model of software functionality that is intended to be reasonable 'on average' for its intended

domain of applicability and uses in project performance measurement and estimating. Caution is therefore needed when measuring, comparing or using sizes of very small pieces of software for these purposes, and especially of very small changes to a piece of software, where the 'average' assumption may break down. In the case of the COSMIC method, 'very small' means 'a few data movements'.

Note: the above advice for 'caution' should not prevent using the COSMIC method in Agile software activities. On the contrary, the method is being used successfully to measure the size of User Stories in Agile software developments, which may have very few data movements. This practice is well-established, with many reports of the CFP sizes of Agile iterations (or 'sprints'), aggregated from the sizes of individual User Stories, that correlate very well with the effort to develop the iteration (and much better than the correlation with effort of sizes measured using Story Points). For an example report on such a comparison, see [16].

4.5.5 Local extension with complex algorithms

If it is judged necessary to account for complex algorithms, a local standard may be arranged for this exceptional functionality. In any functional process where there is an abnormally complex data manipulation functional sub-process, the Measurer is free to assign his or her own locally-determined Function Points.

EXAMPLE: A local extension standard could be: 'In our organization, one Local FP is assigned for mathematical algorithms such as (list of locally meaningful and well-understood examples). Two Local FPs are assigned for (another list of examples), etc.'

4.5.6 Local extension with sub-units of measurement

When more precision is required in the measurement of data movements, then a sub-unit of the measure can be defined. For example, a meter can be sub-divided into 100 centimeters or 1000 millimeters. By analogy, the movement of a single data attribute could be used as a sub-unit of measurement. Measurements on a small sample of software in the field trials of COSMIC indicated that on the software measured, the average number of data attributes per data movement did not vary much across the four types of data movement. For this reason and for ease of measurement reasons the COSMIC unit of measurement, 1 CFP, has been fixed at the level of one data movement. However, caution is clearly needed when comparing the sizes measured in CFP of two different pieces of software where the average number of data attributes per data movement differs sharply across the two pieces of software.

Anyone wishing to refine the COSMIC method by introducing a sub-unit of measurement is free to do so but must make it clear that the resulting size measures are not expressed in standard COSMIC Function Points.

5

MEASUREMENT REPORTING

5.0 Chapter summary

When a measurement is finished and accepted, the result must be reported and data about the measurement archived so as to ensure that the result is always unambiguously interpretable. The chapter lists the parameters that should be considered for recording.

5.1 Labeling

The Generic Software Model can be depicted in matrix form where rows represent functional processes (which might be grouped by layers), columns represent data groups and cells hold the identified sub-processes (Entry, Exit, Read and Write). This representation of the Generic Software Model is presented in Appendix A.

COSMIC measurement results are to be reported and archived according to the following conventions. When reporting a COSMIC functional size it should be labeled according to the following convention, in accordance with the ISO/IEC 14143-1:2007 standard.

RULE – COSMIC measurement labeling
A COSMIC measurement result shall be noted as '*x* CFP (v) ', where: '**x**' represents the numerical value of the functional size,   'v' represents the identification of the version of the standard COSMIC method used to obtain the numerical functional size value 'x'.   NOTE: If a local approximation method was used to obtain the measurement, but otherwise the measurement was made using the conventions of a standard COSMIC version, the above labeling convention shall be used, but use of the approximation method should be noted elsewhere – see section 5.2.

*EXAMPLE: A result obtained using the rules of this Measurement Manual is noted as '**x** CFP (v4.0.2)'*

When local extensions are used, as defined in section 4.5 above, the measurement result must be reported as defined below.

RULE – COSMIC local extensions labeling
A COSMIC measurement result using local extensions shall be noted as: ' *x* CFP (v.) + z Local FP', where: '**x**' represents the numerical value obtained by aggregating all individual measurement results according to the standard COSMIC method, version v,   'v' represents the identification of the version of the standard COSMIC method used to obtain the numerical functional size value 'x'.   'z' represents the numerical value obtained by aggregating all individual measurement results obtained from local extensions to the COSMIC method.

5.2 Archiving COSMIC measurement results

When archiving COSMIC measurement results, the following information should be kept so as to ensure that the result is always interpretable.

RULE – COSMIC measurement reporting

In addition to the actual measurements, recorded as in 5.1, some or all of the following attributes of each measurement should be recorded, depending on the measurement purpose and the desired level of comparability to other measurements, e.g. for benchmarking purposes.

a) Identification of the measured software component (name, version ID or configuration ID).

b) The sources of information used to identify the FUR used for the measurement

c) The domain of the software.

d) A description of the architecture of layers in which the measurement is made, if applicable.

e) A statement of the purpose of the measurement.

f) A description of the scope of the measurement, and its relation to the overall scope of a related set of measurements, if any. (Use the generic scope categories in section 2.2).

g) The measurement (strategy) pattern used (COSMIC or local), with the processing mode (on-line or batch).

h) The functional users of the software.

i) The level of granularity of the available software artefacts and the level of decomposition of the software.

j) The point in the project life-cycle when the measurement was made (especially whether the measurement is an estimate based on incomplete requirements, or was made on the basis of actually delivered functionality).

k) The target or believed error margin of the measurement.

l) Indications whether the standard COSMIC measurement method was used, and/or a local approximation to the standard method, and/or whether local extensions were used (see section 4.5). Use the labeling conventions of sections 5.1 or 5.2.

m) An indication whether the measurement is of developed or delivered functionality ('developed' functionality is obtained by creating new software; 'delivered' functionality includes 'developed' functionality and also includes functionality obtained by other means than creating new software, i.e. including all forms of re-use of existing software, implementation of software packages, use of existing parameters to add or change functionality, etc.).

n) An indication of whether the measurement is of newly provided functionality or is the result of an 'enhancement' activity (i.e. the sum is of added, modified and deleted functionality – see 4.4).

o) The number of major components, if applicable, whose sizes have been added together for the total size recorded.

p) The percentage of functionality implemented by re-used software.

q) For each scope within the overall measurement scope, one measurement matrix, as specified in Appendix A.

---

r) The Measurer's name and any COSMIC certification qualifications; the date of the measurement.

REFERENCES

All the COSMIC documents listed below, including translations into other languages, can be found on www.cosmic-sizing.org .

The COSMIC document titles do not give the method version number to which they relate. All documents are periodically updated to bring them in line with latest version of the method.

[1]   ISO/IEC 19761:2017 Software Engineering – COSMIC: a functional size measurement method, www.iso.org

[2]   Introduction to the COSMIC Method of measuring software

[3]   (Example of several papers by the same authors) Al-Sarayreh, K.T. and A. Abran, Specification and Measurement of System Configuration Non Functional Requirements, 20th International Workshop on Software Measurement (IWSM 2010), Stuttgart, Germany, 2010

[4]   Guideline for Sizing Real-time Software

[5]   Guideline for 'Measurement Strategy Patterns'

[6]   Guideline for early or rapid COSMIC functional size measurement by using approximation approaches

[7]   Guideline for Sizing Business Application Software

[8]   Guideline for sizing Data Warehouse Application Software

[9]   Guideline for Sizing Service-Oriented Architecture Software

[10] Quick Reference Guide to the COSMIC method for sizing Business Application Software

[11] Quick Reference Guide to the COSMIC method for sizing Real-Time Application Software

[12] Guideline for Convertibility (under construction)

[13] International Vocabulary of Basic and General Terms in Metrology, International Organization for Standardization, Switzerland, 2nd edition, 1993, ISBN 92-67-01075-1

[14] Adapted from Merriam Webster's Collegiate Dictionary, 10th Edition

[15] Adapted from Merriam Webster's Collegiate Dictionary, 10th Edition, and La Petit Larousse Illustré, 1996 Edition

[16] Effort Estimation with Story Points and COSMIC Function Points – An Industry Case Study, Christophe Commeyne, Alain Abran, Rachida Djouab, Software Measurement News, Vol 21, No. 1 2016.

[17] ISO/IEC 14143/1:2011 Information technology – software measurement – functional size measurement. Part 1 Definition of concepts

[18] Glossary of terms for 'Non-Functional Requirements and Project Requirements used in software project performance measurement, benchmarking and estimating', COSMIC and IFPUG, September 2015.

[19] See for example: www.wikipedia.org/wiki/AUTOSAR .

APPENDIX A – DOCUMENTING A COSMIC SIZE MEASUREMENT

The structure below can be used as a repository to hold the results of a measurement for each identified software item of an overall scope that has been mapped to the Generic Software Model. Each scope within the overall measurement scope has its own matrix.

The Knowledge base of www.cosmic-sizing.org has a number of spreadsheet tools for measurement recording.

Figure A – Generic Software Model matrix

MEASUREMENT STRATEGY PHASE

- Each piece of software with a defined scope in each layer can be registered as a separate software item

MAPPING PHASE

- Each identified data group is registered in a column

- Each functional process is registered on a specific line, grouped by identified software item.

- For each identified functional process, the identified data movements, whether new or modified are noted in the corresponding cell using the following convention: 'E' for an Entry, 'X' for an Exit, 'R' for a Read and 'W' for a Write

MEASUREMENT PHASE

- For each identified functional process, the data movements are then summed up by type and each total is registered in the appropriate column at the far right of the matrix

- The measurement summary can then be calculated and registered in the boxed cells of each component, on the 'TOTAL' line.

APPENDIX B – EVOLUTION OF NON-FUNCTIONAL REQUIREMENTS – EXAMPLES

The following table lists a few examples of requirements statements that may appear initially at the system level (even before requirements have been allocated to software or hardware) or at the software level as non-functional but that evolve, wholly or partly as a project progresses, into a mixture of FUR for software and statements of requirements that are truly 'non-functional'.

- Column 1: examples of statements of system or software NFR

- Column 2: examples of software FUR that might result, as a project progresses, from the NFR in column 1. The FUR may be for software to be developed or to be acquired e.g. 'COTS' (Commercial Off-the-Shelf) software.

- Column 3: examples of the requirements and constraints on the system or project that might remain after separating out the software FUR in column 2. These are therefore true 'non-functional' requirements.
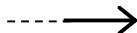
System or software requirements that may initially appear as Non-Functional	Examples of FUR for software, to be developed or acquired, that may evolve from the initial system NFR	Examples of true NFR that may remain after some initial system requirements have evolved into software FUR
The system response time during peak-hour shall not exceed an average of X seconds.	Software to: • Feed external data needed by the system in real-time • Monitor and report on the average response time	• Specific (fast) hardware • Some software to be written in a low-level language • The specific response time target statement
The system availability shall exceed Y% averaged over each calendar year	Software to enable fast switching of processing to a back-up processor without interruption to service	• Back-up hardware processor operating in 'hot standby' mode • The specific availability target statement
Application parameters shall be easily maintainable by user staff	Software to enable users to maintain parameter tables	(None)
The system shall be usable by members of the general public with no training and a Z% successful completion rate	Software to: • provide comprehensive Help facilities • provide well-structured menus for ease of use • support partially-sighted users	• Requirements for Braille keyboards • Extensive testing by members of the general public • The specific Z% target completion rate statement
The user shall have the option to secure files by encryption	Software to encrypt and decrypt files on demand of the user	Use of a hardware 'dongle' or encryption key device
The system shall be portable across X, Y and Z hardware/software environments	A layer of software to isolate the main functionality from the specific interface requirements of the X, Y and Z environments	Use of a highly portable language such as Java

APPENDIX C – CARDINALITY OF TRIGGERING EVENTS, FUNCTIONAL USERS AND FUNCTIONAL PROCESSES

All the relationships along the triggering event / functional user / triggering Entry / functional process chain (as shown in Figure 3.3) may be many-to-many in principle, with one exception. (The exception is that any one triggering Entry may initiate only one functional process – see rule b) for a functional process in section 3.2.2.).

The table below shows examples of possible relationships. Note that the cases may not be exhaustive. The table uses the following abbreviations and symbols.

TEv	Triggering event	FU	Functional User
- - - - ⟶	Data group (dotted part) moved by a triggering Entry (solid arrow)	↓FP	Functional Process

---

1. A single triggering event may cause multiple FUs to each initiate a triggering Entry in the same or in different software systems. Each triggering Entry starts its own FP.
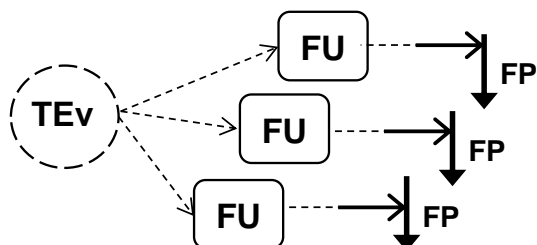
REAL TIME EXAMPLE: The triggering event of an earthquake may be detected by multiple independent FU sensors. Each FU initiates a triggering Entry that starts its FP in the same or in different systems.

BUSINESS EXAMPLE: The triggering event of a new employee starting work causes one human FU to enter basic employee data to a Personnel system and another human FU to enter salary data to a Payroll system.

2. Each triggering event causes a human FU to initiate a different triggering Entry. Each triggering Entry starts its FP in the same or in different software systems.
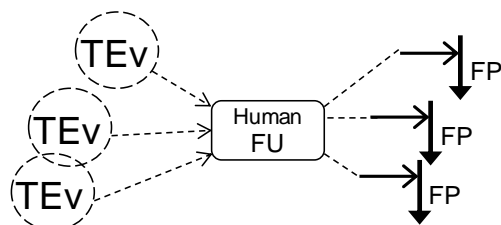
BUSINESS EXAMPLE: In a police emergency telephone call-handling system, many types of triggering events may be reported causing a human FU to decide to initiate different triggering Entries. Each of these starts its FP to record the event. Additionally, the human user may initiate different enquiry triggering Entries. Each of these starts its FP in the same call-handling system or in other systems.

3. A hardware or software FU may be designed to sense (or 'generate') one or more specific type(s) of events. Each of these causes the FU to initiate a triggering Entry. Each of these starts its FP in the same software system.
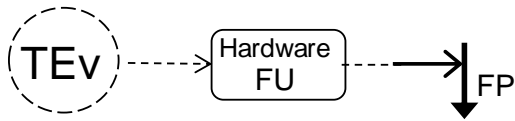
REAL-TIME EXAMPLE: When the temperature of a liquid reaches a pre-set level (the triggering event), a thermocouple FU initiates a triggering Entry to start its FP in one specific software system.
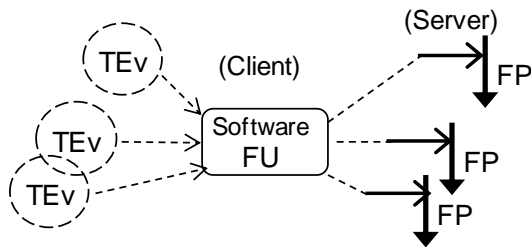
BUSINESS EXAMPLE: In a distributed software application, the client component is a FU of the server component. Different needs for information (the triggering events) of the client component cause it to initiate different triggering Entries, each to start its FP of the server component, for each different type of service it needs.

4. Two or more hardware FUs of the same software may sense the same triggering event. Each FU may initiate the triggering Entry that starts the same one FP.
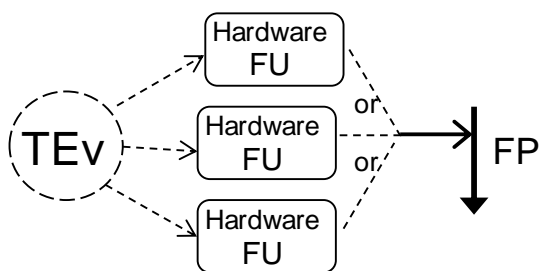
REAL-TIME EXAMPLE: The triggering event of an abnormal situation in a real-time process control system, may be sensed by one or more hardware FUs. Each FU may initiate the one emergency shut-down FP.

(NOTE: Any one occurrence of this FP will be initiated by the first FU to sense the triggering event).

5. Two or more software FUs may each initiate a triggering Entry that starts the same one FP.
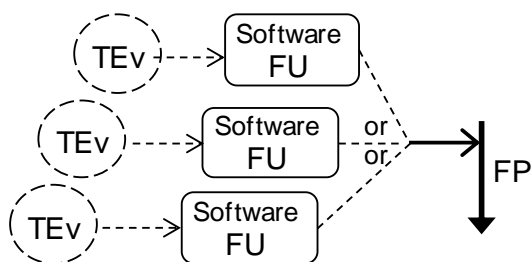
INFRASTRUCTURE EXAMPLE: Several software FUs may each 'call', i.e. initiate, the same FP in the same re-usable software component. (In this case the software FU 'generates' the event when it calls the component.)

(NOTE: Any one occurrence of this FP can be initiated by only one of its possible software FUs at any one time.)

6. On sensing one triggering event, a FU may initiate two or more triggering Entries. Each triggering Entry starts its FP.
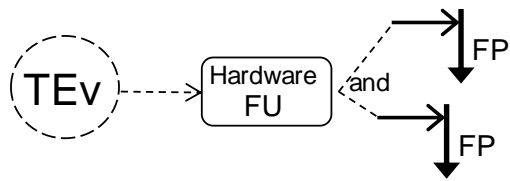
REAL-TIME EXAMPLE: In a duplex safety-critical control system, one triggering event may cause a FU (usually hardware) to initiate two triggering Entries, each starting its FP. The two FPs could, for example, have the same FUR but be developed by separate groups as a result of a diversity strategy.

APPENDIX D – SUMMARY OF COSMIC METHOD PRINCIPLES AND RULES

The table below identifies each principle and rule found in the COSMIC Measurement Method v4.0.2, for the purpose of precise referencing, with the section number in the left column.

Sec.	PRINCIPLES AND RULES DESCRIPTION
1.3.1	**The COSMIC Software Context Model** **Principles** a) Software is bounded by hardware. b) Software is typically structured into **layers**. c) A layer may contain one or more separate **'peer'** pieces of software. d) Any piece of software to be measured, shall be defined by its measurement **scope**, which shall be confined wholly within a single layer. e) The scope of a piece of software to be measured shall depend on the **purpose** of the measurement. f) The **functional users** of a piece of software to be measured shall be identified from its Functional User Requirements (FUR) as the senders and/or intended recipients of data to/from the software respectively. g) The functional requirements of software may be expressed at different **levels of granularity.** h) A precise COSMIC size measurement of a piece of software requires that its FUR are known at the levels of granularity at which its **functional processes** and sub-processes can be identified. i) An approximate COSMIC size measurement of a piece of software is possible if its functional requirements are measured at a high level of granularity by an approximation approach and scaled to the levels of granularity of the functional processes and sub-processes.
1.3.2	**The Generic Software Model** **Principles** a) A piece of software interacts with its functional users across a **boundary**, and with **persistent storage** within this boundary. b) Functional user requirements of a piece of software to be measured can be mapped into unique functional processes. c) Each functional process consists of sub-processes. d) A sub-process may be either a **data movement** or a **data manipulation.** e) A data movement moves a single **data group.** f) There are four data movement types, **Entry**, **Exit**, **Write** and **Read**. • An Entry moves a data group into a functional process from a functional user.

	- An Exit moves a data group out of a functional process to a functional user. - A Write moves a data group from a functional process to persistent storage. - A Read moves a data group from persistent storage to a functional process. g) A data group consists of a unique set of **data attributes** that describe a single **object of interest.** h) Each functional process is started by its **triggering Entry** data movement. The data group moved by the triggering Entry is generated by a functional user in response to a **triggering event.** i) The size of a functional process is equal to the total count of its data movements j) A functional process shall include at least the triggering Entry data movement and either a Write or an Exit data movement, i.e. it shall include a minimum of two data movements. There is no upper limit to the number of data movements in a functional process k) As an approximation for measurement purposes, data manipulation sub-processes are not separately measured; the functionality of any data manipulation is assumed to be accounted for by the data movement with which it is associated NOTE: The COSMIC Generic Software Model, as its name suggests, is a logical 'model' that exposes units in which software processes data that are suitable for functional size measurement. The model does not intend to describe the physical sequence of the steps by which software executes nor any technical implementation of the software.
1.4	**The COSMIC measurement principle** **Principles** a) The size of a functional process is equal to the number of its data movements. b) The functional size of a piece of software of defined scope is equal to the sum of the sizes of its functional processes.
2.2	**Scope of a Measurement** **Rules** a) The scope of any piece of software to be measured shall be derived from the purpose of the measurement. b) The scope of any one measurement shall not extend over more than one layer of the software to be measured
2.2.2	**Layer** **Principles** a) Software in one layer provides a set of services that is cohesive according to some defined criterion, and that software in other layers can utilize without knowing how those services are implemented. b) The relationship between software in any two layers is defined by a 'correspondence rule' which may be either

		• 'hierarchical', i.e. software in layer A is allowed to use the services provided by software in layer B but not vice versa (where the hierarchical relationship may be up or down), or
		• 'bi-directional', i.e. software in layer A is allowed to use software in layer B, and vice versa.
	c)	Software in one layer exchanges data groups with software in another layer via their respective functional processes.
	d)	Software in one layer does not necessarily use all the functional services supplied by software in another layer.
	e)	Software in one layer of a defined software architecture may be partitioned into other layers according to a different defined software architecture.
2.3.1	**Functional users**	
	Rules	
	a)	The functional users of a piece of software to be measured shall depend on the purpose of the measurement.
	b)	When the purpose of a measurement of a piece of software is related to the effort to develop or modify the piece of software, then the functional users should be all the different types of senders and/or intended recipients of data to/from the new or modified functionality, as required by its FUR.
	NOTE: FUR may specify that a set of functional users must be individually identified. Nevertheless they will be of the same type if each occurrence is subject to the same FUR	
2.4.3	**Levels of granularity for measuring a functional process**	
	Rules	
	a)	A functional size measurement of a piece of software requires that its FUR are known at the levels of granularity at which its functional processes and data movement sub-processes may be identified.
	b)	If some requirements must be measured before they have been defined in sufficient detail for a precise measurement, the requirements can be measured using an approximate approach. These approaches define how requirements can be measured at higher level(s) of granularity. Scaling factors are then applied to the measurements at the higher level(s) of granularity to produce an approximate size at the levels of granularity of the functional processes and their data movement sub processes. See the Guideline for Early or Rapid COSMIC Functional Size Measurement by approximation approaches'. [6].
3.2.2	**Functional process**	
	Rules	
	a)	A functional process shall belong entirely to the measurement scope of one piece of software in one, and only one, layer.
	b)	A functional process shall comprise a minimum of two data movements, namely the triggering Entry plus either an Exit or a Write, giving a minimum size of 2 CFP. There is no upper limit to the number of data movements in a functional process and hence no upper limit to its size.
	c)	An executing functional process shall be considered terminated when it has satisfied its FUR for all the possible responses to its triggering Entry. A

		pause during the processing for technical reasons shall not be considered as termination of the functional process.
3.3.1		**Data Group** **Principle** Each identified data group shall be unique and distinguishable through its unique collection of data attributes.
3.3.2		**Identifying different data groups (and hence different objects of interest) moved in the same one functional process** **Rule** For all the data attributes appearing in the input of a functional process: a) sets of data attributes that have different frequencies of occurrence describe different objects of interest; b) sets of data attributes that have the same frequency of occurrence but different identifying key attribute(s) describe different objects of interest; c) all the data attributes in a set resulting from applying parts a) and b) of this rule belong to the same one data group *type*, unless the FUR specify that there may be more than one data group type describing the same object of interest in the input to the functional process (see Note 3) This same rule applies for all the data attributes appearing in the output of a functional process, or all that are moved from a functional process to persistent storage, or all that are moved from persistent storage into a functional process. NOTE 1. It can be helpful when analyzing complex output, e.g. reports with data describing several objects of interest, to consider each separate candidate data group as if it were output by one separate functional process. Each of the data group types identified this way must also be distinguished and counted when measuring the complex report. For examples, see the 'Guideline for sizing business application software' [7], in particular the example in section 2.6.1 and its analysis in 2.6.2. See also the analysis of the examples 4 and 5 in section 4.2.4. NOTE 2. Examining how data attributes are physically grouped or separated on *input or output* may help distinguish different data group types, but cannot be relied upon to distinguish them. As an example, two or more sets of data attributes occurring on the same input or output that are physically separated for aesthetic reasons or for ease of understanding, will belong to the same one data group type if they satisfy the rule above. NOTE 3. See section 3.5 of the Measurement Manual for the definitions, principles and rules for the *data movements* that move data groups, and section 3.5.7 (examples 2, 3, 4 and 5) and 3.5.11 for exceptions to these rules for data movements, as per rule c above.
3.5.2		**Entry (E)** **Principles** a) An Entry shall move a single data group describing a single object of interest from a functional user across the boundary and into the functional process of which the Entry forms part. If the input to a functional process comprises more than one data group, each describing a different object of interest, identify one Entry for each unique data group in the input. (See also section 3.5.7 on 'Data movement uniqueness'.)

	b) An Entry shall not exit data across the boundary, or read or write data from/to persistent storage.
	Rules
	a) The data group of a triggering Entry may consist of only one data attribute which simply informs the software that 'an event Y has occurred'. Very often, especially in business application software, the data group of the triggering Entry has several data attributes which inform the software that 'an event Y has occurred and here is the data about that particular event'.
	b) Clock-ticks that are triggering events shall always be external to the software being measured. Therefore, for example, a clock-tick event occurring every 3 seconds shall be associated with an Entry moving a data group of one data attribute. Note that it makes no difference whether the triggering event is generated periodically by hardware or by another piece of software outside of the boundary of the software being measured.
	c) Unless a specific functional process is necessary, obtaining the date and/or time from the system's clock shall not be considered to cause an Entry or any other data movement.
	d) If an occurrence of a specific event causes the Entry of a data group comprising up to 'n' data attributes of a particular object of interest and the FUR allows that other occurrences of the same event can cause an Entry of a data group which has values for attributes of only a sub-set of the 'n' attributes of the object of interest, then one Entry shall be identified, moving a data group comprising all 'n' data attributes.
	e) When identifying Entries in a screen that enables human functional users to input data into functional processes, analyze only screens that are filled with data. Ignore any screen that is formatted but otherwise 'blank' except for possible default values, and ignore all field and other headings that enable human users to understand the input data required.
	NOTE. It may be necessary to consider field and other headings when measuring FUR for changes to Entries – see section 4.4.1.
3.5.3	**Exit (X)**
	Principles
	a) An Exit shall move a single data group describing a single object of interest from the functional process of which the Exit forms part across the boundary to a functional user. If the output of a functional process comprises more than one data group, identify one Exit for each unique data group in the output. (See also section 3.5.7 on 'Data movement uniqueness'.)
	b) An Exit shall not enter data across the boundary, or read or write data from/to persistent storage.
	Rules
	a) An enquiry which outputs fixed text, (where 'fixed' means the message contains no variable data values e.g. the result of pressing a button for 'Terms & Conditions' on a shopping web-site), shall be modeled as having one Exit for the fixed text output.
	NOTE: For the output from 'Help' functionality, see the 'Guideline for sizing Business Application Software'. For the output of messages concerned with error conditions or confirming success, see section 3.5.11 of this Measurement Manual.

	b)	If an Exit of a functional process moves a data group comprising up to 'n' data attributes of a particular object of interest and the FUR allows that the functional process may have an occurrence of an Exit that moves a data group which has values for attributes of only a sub-set of the 'n' attributes of the object of interest, then one Exit shall be identified, moving a data group comprising all 'n' data attributes.
	c)	When identifying Exits, ignore all field and other headings that enable human users to understand the output data. NOTE: It may be necessary to consider field and other headings when measuring FUR for changes to Exits – see section 4.4.1
3.5.4	**Read (R)** **Principles**	
	a)	A Read shall move a single data group describing a single object of interest from persistent storage to a functional process of which the Read forms part. If the functional process must retrieve more than one data group from persistent storage, identify one Read for each unique data group that is retrieved. (See also section 3.5.7 on 'Data movement uniqueness'.)
	b)	A Read shall not receive or exit data across the boundary or write data to persistent storage.
	c)	During a functional process, movement or manipulation of constants or variables which are internal to the functional process and that can be changed only by a programmer, or computation of intermediate results in a calculation, or of data stored by a functional process resulting only from the implementation, rather than from the FUR, shall not be considered as Read data movements.
	d)	A Read data movement always includes any 'request to Read' functionality (so a separate data movement shall never be counted for any 'request to Read' functionality). See also section 3.5.9.
	Rules	
	a)	Identify a Read when, according to the FUR, the software being measured must retrieve a data group from persistent storage.
	b)	Do not identify a Read when the FUR of the software being measured specify any software or hardware functional user as the source of a data group, or as the means of retrieving a persistently-stored data group. (For this case see the principles and rules for Entries and Exits.)
3.5.5	**Write (W)** **Principles**	
	a)	A Write shall move a single data group describing a single object of interest from the functional process of which the Write forms part to persistent storage. If the functional process must move more than one data group to persistent storage, identify one Write for each unique data group that is moved to persistent storage. (See also section 3.5.7 on 'Data movement uniqueness'.)
	b)	A Write shall not receive or exit data across the boundary, or read data from persistent storage.
	c)	A requirement to delete a data group from persistent storage shall be measured as a single Write data movement.

	d) The following shall not be considered as Write data movements:
	• The movement or manipulation of any data that did not exist at the start of a functional process and that has not been made persistent when the functional process is complete;
	• Creation or update of variables or intermediate results that are internal to the functional process;
	• Storage of data by a functional process resulting only from the implementation, rather than from the FUR. (An example would be the use of storage to store data temporarily during a large sort process in a batch-processed job.)
	Rules
	a) Identify a Write when, according to the FUR, the software being measured must move a data group to persistent storage.
	b) Do not identify a Write when the FUR of the software being measured specify any software or hardware functional user as the destination of the data group, or as the means of storing the data group. (For this case see the principles and rules for Entries and Exits.)
3.5.6	**Data manipulation associated with data movements**
	Principle
	All data manipulation in a functional process shall be associated with the four types of data movement (E, X, R, and W). By convention, the data movements of a functional process are assumed also to account for the data manipulation of the functional process.
	Rules
	a) An Entry data movement accounts for all data manipulation to enable a data group to be entered by a functional user (e.g. formatting and presentation manipulations) and to be validated.
	b) An Exit data movement accounts for all data manipulation to create the data attributes of a data group to be output and/or to enable the data group to be output (e.g. formatting and presentation manipulations) and to be routed to the intended functional user.
	c) A Read data movement accounts for all computation and/or logical processing needed in order to retrieve a data group from persistent storage.
	d) A Write data movement accounts for all computation and/or logical processing to create or to update a data group to be written, or to delete a data group.
	e) The data manipulation associated with any of these data movements does not include any data manipulation that is needed after the data movement has been successfully completed, nor does it include any data manipulation associated with any other data movement.
3.5.7	**Data movement uniqueness and possible exceptions**
	Rules
	N.B. All COSMIC rules concern types of functional users, data groups, data movements, functional processes and objects of interest. For ease of reading, we normally omit 'type' from these terms. This convention is followed in rules a), b) and c) below but In rule d) we include 'type' where it is helpful to distinguish a

		'type' from an 'occurrence'.
		a) Unless the Functional User Requirements are as given in rules b) or c), all data describing any one object of interest that is required to be entered into one functional process shall be identified as one data group moved by one Entry.
		NOTE: A functional process may, of course, have multiple Entries, each moving data describing a different object of interest.
		The same equivalent rule applies to any Read, Write or Exit data movement in any one functional process.
		b) AIf Functional User Requirements specify that different data groups must be entered into one functional process each from a different functional user, where each data group describes the same object of interest then one Entry shall be identified for each of these different data groups.
		The same equivalent rule applies for Exits of data to different functional users from any one functional process.
		NOTE: Any one functional process shall have only one triggering Entry.
		c) If Functional User Requirements specify that different data groups must be moved from persistent storage into one functional process, each describing the same object of interest then one Read shall be identified for each of these different data groups.
		The same equivalent rule applies for Writes in any given functional process.
		NOTE: This rule is analogous to rule b). In the case of the FUR to read different data groups desrbing the same object of interest, they will likely have originated from different functional users. In the case of the FUR to write different data groups, they will likely be made available to be read by different functional users.
		d) Repeated occurrences of any data movement type when it is being executed shall not be counted.
		This applies even if multiple occurrences of the data movement type differ in their execution because different values of the data attributes of the data group moved result in different processing paths being followed through the functional process type.
3.5.9	**A functional process requiring data from a functional user** **Rules**	
		a) A functional process shall obtain a data group via an Entry data movement from a functional user, when the functional process does not need to tell the functional user what data to send, as in any of the following four cases:
		• when a functional user sends a data group via a triggering Entry which initiates the functional process;
		• when a functional process, having received a data group via a triggering Entry, waits, expecting the arrival of a further data group from the functional user via an Entry (can occur when a human functional user enters data to business application software);
		• when a functional process, having started, requests the functional user, 'send me your data now, if you have any' and the functional user sends its data;

	• when a functional process, having started, inspects the state of a functional user and retrieves the data it requires.
	In the latter two cases (typically occurring in real-time 'polling' software), by convention no Exit from the functional process shall be identified to obtain the required data. The functional process merely needs to send a prompt message to a functional user and the functionality of that prompt message is considered to be part of the Entry. The functional process knows what data to expect. Only one Entry shall be identified for this case.
	b) Where a functional process needs to obtain the services of a functional user (for instance to obtain data) and *the functional user needs to be told what to send* (typically where the functional user is another piece of software outside the scope of the software being measured), an Exit followed by an Entry data movement shall be identified. The Exit sends the request for the specific data; the Entry receives the returned data.
3.5.10	**Control commands in applications with a human interface**
	Rule
	In an application with a human interface 'control commands' shall be ignored as they do not involve any movement of data about an object of interest.
3.5.11	**Error/Confirmation messages and other indications of error conditions**
	Rules
	a) One Exit shall be identified to account for all types of error/confirmation messages issued by any one functional process of the software being measured from all possible causes according to its FUR, e.g. successes or failures of: validation of entered data or for a call to retrieve data or to make data persistent, or for the response from a service requested of another piece of software.
	NOTE: If the FUR of the functional process do not require any type of error/confirmation message to be issued, do not identify any corresponding Exit.
	b) If a message to a <u>human</u> functional user provides data in addition to confirming that entered data has been accepted, or that entered data is in error, then this additional data should be identified as a data group moved by an Exit in the normal way, in addition to the error/confirmation Exit.
	c) All other data, issued or received by the software being measured, to/from its <u>hardware or software functional users</u> should be analyzed according to the FUR as Exits or Entries respectively, according to the normal COSMIC rules, regardless of whether or not the data values indicate an error condition.
	d) Reads and Writes are considered to account for any associated reporting of error conditions. Therefore no Entry to the functional process being measured shall be identified for any error indication received as a result of a Read or Write of persistent data.
	e) No Entry or Exit shall be identified for any message indicating an error condition that might be issued whilst using the software being measured but which is not required to be processed in any way by the FUR of that software, e.g. an error message issued by the operating system.
4.3.1	**Aggregating measurement results**
	Rules

	a) For any functional process, the functional sizes of individual data movements shall be aggregated into a single functional size value in units of CFP by adding them together. $$\text{Size (functional process}_i) = \Sigma \text{ size(Entries)} + \Sigma \text{ size(Exits)} +$$ $$\Sigma \text{ size(Reads)} + \Sigma \text{ size(Writes)}$$ b) For any functional process, the functional size of changes to its Functional User Requirements shall be aggregated from the sizes of the data movements that have been added, modified or deleted in the functional process to give a size of the change in units of CFP, according to the following formula. $$\text{Size (Change(functional process}_i)) =$$ $$\Sigma \text{ size (added data movements)} +$$ $$\Sigma \text{ size (modified data movements)} +$$ $$\Sigma \text{ size (deleted data movements)}$$ For more on aggregating functional size, see section 4.3.2. For measuring the size of the changed software, see section 4.4.2. c) The size of a piece of software within a defined scope shall be obtained by aggregating the sizes of the functional processes for the piece, subject to rules e) and f) below d) The size of any change to a piece of software within a defined scope shall be obtained by aggregating the sizes of all changes to all functional processes for the piece, subject to rules e) and f) below. e) Sizes of pieces of software or of changes to pieces of software may be added together only if measured at the same functional process level of granularity of their FUR. f) Sizes of pieces of software and/or changes in the sizes of pieces of software within any one layer or from different layers shall be added together only if it makes sense to do so, for the purpose of the measurement. g) The size of a piece of software may be obtained by adding up the sizes of its components (regardless of how the piece is decomposed) and eliminating the size contributions of inter-component data movements. h) Only one Exit shall be identified for all error/confirmation messages issued by any one functional process to a human functional user. i) If the COSMIC method is extended locally (for example to measure some aspect of size not covered by the standard method), then the size measured via the local extension must be reported separately as described in section 5.1 and shall NOT be added to the size obtained by the standard method, measured in CFP (see further in section 4.5).
4.4.1	**Modifying a data movement** **Rules** a) If a data movement must be modified due to a change of the data manipulation associated with the data movement and/or due to a change in the number or type of the attributes in the data group moved, one changed CFP shall be measured, regardless of the actual number of modifications in

	the one data movement.
	b) If a data group must be modified, data movements moving the modified data group whose functionality is not affected by the modification to the data group shall not be identified as changed data movements.
	NOTE: A modification to any data appearing on input or output screens that are not related to an object of interest to a functional user shall not be identified as a changed CFP. (See section 3.3.3 for examples of such data.)
4.4.2	**Size of functionally-changed software**
	Rule
	After functionally changing a piece of software:
	New total size (changed piece of software) = Old total size (piece of software)
	$+\quad \Sigma$ size (added data movements)
	$-\quad \Sigma$ size (deleted data movements)
5.1	**COSMIC measurement labeling**
	Rules
	A COSMIC measurement result shall be noted as '*x* CFP (v) ', where:
	• '**x**' represents the numerical value of the functional size,
	• 'v' represents the identification of the standard version of the COSMIC method used to obtain the numerical functional size value 'x'.
	NOTE: If a local approximation method was used to obtain the measurement, but otherwise the measurement was made using the conventions of a standard COSMIC version, the above labeling convention shall be used, but use of the approximation method should be noted elsewhere – see section 5.2.
5.1	**COSMIC local extensions labeling**
	Rule
	A COSMIC measurement result using local extensions shall be noted as:
	' *x* CFP (v) + z Local FP', where:
	• '**x**' represents the numerical value obtained by aggregating all individual measurement results according to the standard COSMIC method, version v,
	• 'v' represents the identification of the standard version of the COSMIC method used to obtain the numerical functional size value 'x'.
	• 'z' represents the numerical value obtained by aggregating all individual measurement results obtained from local extensions to the COSMIC method.
5.2	**COSMIC measurement reporting**
	Rule
	In addition to the actual measurements, recorded as in 5.1, some or all of the following attributes of each measurement should be recorded, depending on the measurement purpose and the desired level of comparability to other measurements, e.g. for benchmarking purposes.
	a) Identification of the measured software component (name, version ID or configuration ID).
	b) The sources of information used to identify the FUR used for the measurement

	c)	The domain of the software
	d)	A description of the architecture of layers in which the measurement is made, if applicable.
	e)	A statement of the purpose of the measurement.
	f)	A description of the scope of the measurement, and its relation to the overall scope of a related set of measurements, if any. (Use the generic scope categories in section 2.2)
	g)	The measurement strategy pattern used (COSMIC or local), with the processing mode (on-line or batch)
	h)	The functional users of the software
	i)	The level of granularity of the available software artefacts and the level of decomposition of the software.
	j)	The point in the project life-cycle when the measurement was made (especially whether the measurement is an estimate based on incomplete requirements, or was made on the basis of actually delivered functionality).
	k)	The target or believed error margin of the measurement.
	l)	Indications whether the standard COSMIC measurement method was used, and/or a local approximation to the standard method, and/or whether local extensions were used (see section 4.5). Use the labeling conventions of sections 5.1 or 5.2.
	m)	An indication whether the measurement is of developed or delivered functionality ('developed' functionality is obtained by creating new software; 'delivered' functionality includes 'developed' functionality and also includes functionality obtained by other means than creating new software, i.e. including all forms of re-use of existing software, implementation of software packages, use of existing parameters to add or change functionality, etc.).
	n)	An indication of whether the measurement is of newly provided functionality or is the result of an 'enhancement' activity (i.e. the sum is of added, changed and deleted functionality – see 4.4).
	o)	The number of major components, if applicable, whose sizes have been added together for the total size recorded.
	p)	The percentage of functionality implemented by re-used software
	q)	For each scope within the overall measurement scope, one measurement matrix, as specified in Appendix A.
	r)	The Measurer's name and any COSMIC certification qualifications; the date of the measurement

APPENDIX E – THE MAIN CHANGES FROM VERSION 4.0 TO VERSIONS V4.0.1 AND 4.0.2

This Appendix contains a summary of the principal changes made in the evolution of the COSMIC functional size measurement method from version 4.0 to v4.0.1 and to the present v4.0.2.

To trace the earlier evolution of the method, please refer to the Measurement Manual for each earlier version of the method (2.2, 3.0 and 3.0.1, and 4.0).

A 'MUB' is a Method Update Bulletin, published between major releases of the Measurement Manual to announce and explain proposed changes.

E1: Main changes from v4.0 to v4.0.1

V4.0.1 Ref	Change
Foreword	The Introduction document should be read first if you are a beginner or are converting, this has been emphasized by adding a title to the existing paragraph
1.2.3	The definition of a 'Non-Functional Requirement' has been made clearer and does not now include project requirements and constraints. Figure 1.3 has been changed correspondingly.
1.3.3	A new section 1.3.3 'Types versus occurrences' has been added. COSMIC has never defined 'type' and 'occurrence' explicitly and it appeared that Measurers still sometimes have difficulty in understanding the difference,
3.2.1	The definition of 'functional process' was ambiguous at one point as it was not clear whether 'unique' refers to 'the set of data movements' or to 'the elementary part of the FUR'. It has been solved by adding a comma after 'movements' and changing two occurrences of 'that' to 'these'.
3.3.1	The definition for a 'data group' has been simplified as it was unnecessarily complex. Two of the principles have been removed as they as they did not add value.
3.3.1	The definition of an 'object of interest' has been revised and a note added, both aiming to make the definition clearer
3.5.1	The definitions of a 'data movement', 'Entry', 'Exit', 'Read' and 'Write' now state that each is considered to 'account for' (not 'include') any associated data manipulation
3.5.6	The text now emphasizes that the rules in this section are relevant only to the measurement of changes to existing FUR.
	The rules that define which types of data manipulations are associated with which data movements now make clear that data manipulation is 'associated with', not 'included' in data movements.
3.5.7	The rules for the 'Data movement uniqueness and possible exceptions' have been reworded as they were complex. An error has been corrected (two data movements cannot be considered as different due to their having different FUR

V4.0.1 Ref	Change
	for their associated data manipulation as this would contradict the principle that 'all data manipulation in a functional process shall be associated with the four types of data movement (E, X, R and W)'). Two examples have been added and the examples have been re-sequenced to align with the revised rules. Some editorial simplifications have also been made.
3.5.11	The definition of 'error/confirmation message' has been clarified. Part of the previous definition has been made a rule.
4.3.1	The rule g) for how to obtain the size of a piece of software from the sizes of its components has been changed from a 'double-negative' to a positive statement. The rule has been extended on how to aggregate error/confirmation messages.
Various	References to the ISO/IEC standards 14143/1 on the Definition of concepts (of FSM methods) and to ISO/IEC 19761 which defines the main concepts of the COSMIC method were confused and have now been corrected.
Glossary	The definitions of 'input' and 'output' have been simplified as they were unnecessarily complex.

E2: Main changes from v4.0.1 to v4.0.2

Note. The nature of a change is indicated by:-

- 'Method' when a definition or rule of the COSMIC method has been added or has been changed to improve understanding. (Changes to, or additions of, Notes are considered as 'Editorial' changes.)

- 'Editorial' when some text has been changed to improve ease of understanding. Many minor editorial improvements have also been made in addition to those listed below.

- 'Correction' when an error in the previous version v4.0.1 of this Guideline has been corrected.

V4.0.2 Ref	Nature of change	Change
-	Editorial	The provisional name 'Guideline for approximate COSMIC functional size measurement' has been changed to its definitive name 'Guideline for Early or Rapid COSMIC Functional Size Measurement by approximation approaches'.
Foreword		A summary of the main changes for v4.0.2 has been added.
1.2	Editorial	In version 4.0 of the method, we restricted how we would use the term 'Functional User Requirements' (FUR) in the COSMIC method. The restriction has now been added as a Note 2 to the term's definition.
1.2, 1.3	Corrections	In several places in Chapter 1, notably in two Principles of the Software Context Model, the term 'FUR' has been replaced by 'functional requirements' to reflect the now-restricted use of 'FUR'. These changes should have been made when the restricted interpretation was introduced in v4.0.
1.3.2	Method	A Principle has been added stating 'The size of a functional process is equal to the total count of its data movements'. This

V4.0.2 Ref	Nature of change	Change
	Editorial	has only ever been stated as a rule in section 4.3.1. The Principle concerning the size range of a functional process now states explicitly that there is no upper limit to its size. A Note has been added to the Generic Software Model explaining that the model does not intend to describe the physical sequence of steps by which a functional process executes.
1.3.3	Editorial	This section has been re-structured and the explanation of the term 'type' has been improved to help understanding, in particular to help identify when functional users are of the same type. The explanation of Real-time example 2 has also been improved.
2.0	Editorial	The Chapter Summary has been improved to be simpler and to better align with the Chapter content.
2.2.1	Editorial	The text of the Business Example illustrated by Figure 2.1 has been improved for clarity.
2.2.2	Editorial	The obsolete www reference to the AUTOSAR architecture has been replaced.
2.2.3	Editorial	In Note 2 to the definition of 'Levels of decomposition', the phrase 'may only be' has been changed to 'are only', as use of 'may be' is not correct in this context. A Note 3 has been added that levels of decomposition may correspond to layers of the software's architecture, but not necessarily.
2.3	Editorial	The section title has been changed because 'Persistent storage' does not need to be identified. The opening part of 2.3 and the section 2.3.1 have been re-structured into a more logical sequence.
2.3.1	Method	The definition of 'functional user' has been made more specific (MUB 12).
2.3.1	Editorial	A Note has been added to the rules for a 'functional user' to help understanding. Example 1 as presented in v4.0.1 was not really an example. It has been re-written as text, and the following examples have been re-numbered. Further explanations of 'functional user' have been added. Real-time Example 2 has been 'modernized' by changing the subject from a mobile phone with a keyboard (now less common) to a copier. Real-time Example 4 has been 'modernized' so that the output is to a dashboard display screen, rather than to four LEDs.
2.3.2	Editorial	Further explanation of persistent storage has been added and its relationship to files or databases.

V4.0.2 Ref	Nature of change	Change
2.4.1	Method	The definition of 'Level of Granularity' has been expanded slightly to make clear that it applies to the description of 'any part of' a piece of software, so not necessarily to the whole description.
2.4.3	Correction	The definition of 'Functional process level of granularity' has been improved for ease of understanding, including correction for the now-restricted use of the term 'FUR'. The rules for 'Functional process level of granularity' have been re-named as 'Levels of granularity for measuring a functional process', to reflect the facts that measuring a functional process requires FUR at two levels of granularity, namely that of the functional process and of its data movements, and that rule b) also concerns approximate measurement at higher levels of granularity.
2.4.3	Editorial	It is made clear that the whole 'Everest Ordering System' example pertains to the level of granularity of the FUR (not to the level of decomposition of the software). The purpose of the examples has also been made clearer.
3.2.1	Method	The definition of 'Triggering event' has been modified to make clear that only the first data group generated by any one functional user will be moved by a triggering Entry (MUB 14).
3.2.1	Method	In the second bullet point of the definition of a Functional process, 'may' has been replaced by 'shall'. 'May' is technically correct because, as explained in the Foreword, in ISO terminology 'may' means 'is allowed to'. However using 'shall' makes it clearer that this is a mandatory part of the definition. A Note 4 has been added to the definition to explain the use of 'unique' in clause a) of the definition.
3.2.1	Editorial	In the definition of 'Triggering Entry, a Note has been added to explain that the phrase 'that the functional process needs to start processing' is the result of the logical GSM and does not necessarily mean that processing physically starts when data is entered. In the definition of 'Triggering Entry' in the Glossary, a Note has been deleted. This Note really belongs in the definition of a Functional Process, where it is correctly stated.
3.2.1	Editorial	In Figure 3.3, the Data group arrow has been replaced by a parallelogram symbol to give it equal prominence with the other symbols on the Figure.
3.2.2	Editorial	Rule b) for a functional process has been removed, as it is superfluous. The wording of (the new) rule b) has been improved for ease of understanding. The wording of (the new) rule c) has been changed slightly to bring it in line with the definition of a functional process.

V4.0.2 Ref	Nature of change	Change
3.2.3	Editorial	The description of the measurement of batch-processed functional processes has been expanded to acknowledge that input data may arrive as a transmitted data stream (as well as via a temporarily-stored dataset). Figure 3.4 shows the change as well.
3.3.1	Method	In the definition of object of interest the phrase 'process and/or move data' has been replaced by 'move a data group in or out of the software, or to or from persistent storage'. The change aims to prevent a possible interpretation that objects of interests may be identified solely due to 'processing', i.e. data manipulation. As a consequence, Note 1 pertaining to 'processing' is no longer needed and has been removed. A Note 2 has been added that when a functional user sends data about itself, the functional user is also the object of interest of the data group sent. A Note 3 has been added that there is nothing absolute about whether a 'thing' is an 'object of interest' or not (MUB 12).
3.3.1	Editorial	A Note has been added to the definition of a 'data group', clarifying that it need not consist of all the data attributes of an object of interest (MUB 13).
3.3.1	Editorial	The definition of 'Data attribute' has been moved from 3.4.1 to 3.3.1, where it more logically belongs.
3.3.2	Method	A new rule has been added: 'Identifying different data groups (and hence different objects of interest) moved in the same one functional process'. The new rule aims in particular to help identify data groups moved in complex output from business application functional processes. However, the rule is valid for software from all domains and for input and data moved to or from persistent storage.
3.3.2	Editorial	Almost the whole of this section has been re-written and more examples have been added to make the text clearer.
3.3.4	Correction	The text has been revised to make clear that a functional user can be an object of interest in the business application domain as well as the real-time domain., A business example has been added (in line with MUB 12).
3.4.1	Editorial	This section has been renamed 'Data attribute examples' and the examples have been much improved
3.5.2	Method	Rule c) for an Entry has been expanded to make clear that an Entry is not needed to obtain either the current date and/or the time from the operating system. In rule d) for an Entry. The word 'triggers' has been replaced by 'causes', to avoid a mis-interpretation that this rule applies only to a triggering Entry
3.5.3	Editorial	Rule b) for an Exit has been modified slightly for ease of

V4.0.2 Ref	Nature of change	Change
		understanding.
3.5.6	Correction	The definition of data manipulation 'anything that happens to data .... other than movement ...' has been modified to make clear that this applies only to processing by (or 'within') a functional process.
3.5.7	Editorial	The statement in the opening paragraph that the 'The Generic Software Model assumes that normally ...' is wrong and has been deleted. It could imply that this Model allows exceptions, which is not true. The paragraph now begins 'Most often ....' Some wording changes have been made to the 'Data Movement Uniqueness' rules and a Note has been added to improve ease of understanding. In particular, the last sentence of clause d) has been deleted to avoid potential confusion. The sentence is not actually incorrect but may appear to conflict with the new rules for distinguishing data groups and objects of interest (see section 3.3.2). Example 10 has been changed for clarity; it is relevant only to rule d)
3.5.10	Editorial	Examples have been removed from the definition of 'Control Commands'. The examples have been rationalized and re-written for greater clarity.
3.5.8	Editorial	A paragraph has been added to explain that the Measurer must first decide whether the retrieval of data must take place from persistent storage within the boundary of the software being measured or from persistent storage accessible via another piece of software.
3.5.11	Editorial	Business Example 3 has been re-written to improve ease of understanding.
4.3.1	Method	The two parts of rule g) for the aggregation of measurement results has been split into two rules, as the two parts are unrelated.
4.4.2	Method	The equation for measuring the size of functionally-changed software has been changed from text to a Rule, and an Example has been added.
4.5.4	Editorial	A Note has been added to describe that COSMIC Functional Size Measurement has been successfully applied to the measurements of User Stories and sprints in Agile developments.
App. C	Correction	Example 6) isn't limited to 'the same software', this phrase has been removed.

APPENDIX F - GLOSSARY OF TERMS

The following terms are used throughout the COSMIC functional size measurement method (the 'COSMIC method'), according to the definitions found in this section. Terms already defined by ISO, such as 'Functional Size Measurement' or 'unit of measurement', together with their ISO definition have also been adopted for the COSMIC method.

For many of the terms listed in the glossary, when appropriate, the suffix 'type' is shown. Since any functional size measurement method aims to identify 'types' and not 'occurrences' of data or functions, almost invariably throughout the COSMIC method we will be concerned with 'types' and not 'occurrences'. Consequently, in the texts we will drop the suffix 'type' from these terms for the sake of readability, except when we specifically need to distinguish type and occurrence. This is also the convention adopted in the International Standard (ISO/IEC 19761:2011) definition of the COSMIC method. Occasionally this convention leads to difficulties when drafting these definitions – see for example Note 3 of the definition of 'data movement type' below, which does not appear in the International Standard.

For a fuller discussion of 'type' versus 'occurrence', see section 1.3.3.

NOTE: Terms that are used only in the domain-specific COSMIC 'guidelines' are defined in those guidelines; they are not shown below.

In the definitions given below:

- terms that are defined elsewhere in this glossary are under-lined, for ease of cross-reference.

- terms that originate in the ISO Standard for the COSMIC method (ISO/IEC 19761) or that are otherwise specific to the COSMIC method are shown in ***bold italics***.

- other terms that have been adopted from ISO but that are not specific to the COSMIC method are shown in **bold**.

Application. A software system for collecting, saving, processing, and presenting data by means of a computer.

NOTE: This is an adaptation of the definition given in ISO/IEC 24570:2005 Software engineering -- NESMA functional size measurement method version 2.1.

(Alternative definition for 'application software'). Software designed to help users perform particular tasks or to handle particular types of problems, as distinct from software that controls the computer itself.

NOTE: This is a slight adaptation of the definition given in ISO/IEC 24765:2010 Systems and software engineering-vocabulary, 4.5).

Application-general data. Any data related to the application in general and not related to an object of interest of a specific functional process.

Base functional component (BFC). An elementary unit of the Functional User Requirements defined by an FSM method for measurement purposes [17].

NOTE: The COSMIC method defines a data movement type as a BFC.

Base functional component type (BFC type). A defined category of BFCs [17]. The COSMIC method has four BFC types, the Entry, Exit, Read and Write (-types).

Boundary. A conceptual interface between the <u>software</u> being measured and its <u>functional users</u>.

NOTE: It follows from the definition that there is a boundary between any two pieces of software in the same or different <u>layers</u> that exchange data where one <u>piece of software</u> is a <u>functional user</u> of the other, and/or vice versa.

Component. Any part of a <u>software system</u> that is separate for reasons of the software architecture, and/or that it was specified, designed or developed separately.

Control command. A command that enables human <u>functional users</u> to control their use of the software but which does not involve any movement of data about an <u>object of interest</u> of the FUR of the software being measured.

NOTE: A <u>control command</u> is not a <u>data movement</u> because the command does not move data about an <u>object of interest</u>.

COSMIC unit of measurement. 1 CFP (<u>C</u>osmic <u>F</u>unction <u>P</u>oint), which is defined as the size of one <u>data movement</u>

NOTE: The unit of measurement was known as a 'Cfsu' (COSMIC functional size unit) prior to v3.0 of the method.

Data attribute type (synonym 'data element type'). The smallest parcel of information, within an identified <u>data group</u> type, carrying a meaning from the perspective of the software's <u>Functional User Requirements</u>.

Data group type. A distinct, non-empty and non-ordered set of <u>data attribute</u> types where each included <u>data attribute</u> type describes a complementary aspect of the same one <u>object of interest</u>.

NOTE: 'Data group' does not necessarily mean 'the set of *all* data attributes that describe a single object of interest'. The FUR of a piece of software may specify data groups to be formed from any combinations of data attributes that all describe the same object of interest, as needed by different functional processes.

Data manipulation. Anything that happens to data when it is processed by a functional process other than a movement of data into or out of a <u>functional process</u>, or between a <u>functional process</u> and <u>persistent storage</u>.

Data movement type. A <u>base functional component</u> which moves a single <u>data group</u> type.

NOTE 1: There are four sub-types of a data movement type, namely: <u>Entry</u>, <u>Exit</u>, <u>Read</u> and <u>Write</u> (-types).

NOTE 2: For measurement purposes, each data movement is considered to account for certain associated <u>data manipulation</u> – see the Measurement Manual for details.

NOTE 3: More precisely, it is an *occurrence* of a data movement, not a data movement *type*, that actually *moves* the <u>data group</u> *occurrences* (not *types*). This comment also applies to the definitions of <u>Entry</u>, <u>Exit</u>, <u>Read</u> and <u>Write</u>.

E. Abbreviation for '<u>Entry</u> type'.

Entry type. A <u>data movement</u> that moves a <u>data group</u> from a <u>functional user</u> across the <u>boundary</u> into the <u>functional process</u> where it is required**.**

NOTE: An Entry type is considered to account for certain associated <u>data manipulation</u> – see the Measurement Manual for details.

Error/confirmation message. An <u>Exit</u> issued by a <u>functional process</u> to a human user that either confirms only that entered data has been accepted, or only that there is an error in the entered data.

NOTE: Any Exit that may include fault indications, but that is not intended for a human functional user, is not an error/confirmation message,

Event type. Something that happens.

Exit type. A data movement that moves a data group from a functional process across the boundary to the functional user that requires it.

NOTE: An Exit type is considered to account for certain associated data manipulation – see the Measurement Manual for details.

Functional process type

a) A set of data movements, representing an elementary part of the Functional User Requirements for the software being measured, that is unique within these FUR and that can be defined independently of any other functional process in these FUR

b) A functional process shall have only one triggering Entry. Each functional process starts processing on receipt of a data group moved by the triggering Entry data movement of the functional process.

c) The set of all data movements of a functional process is the set that is needed to meet its FUR for all the possible responses to its triggering Entry.

NOTE 1: When implemented, it is an *occurrence* of a functional process that starts *executing* on receipt of an *occurrence* of a data group moved by an *occurrence* of a triggering Entry.

NOTE 2: The FUR for a functional process may require one or more other Entries in addition to the triggering Entry.

NOTE 3: If a functional user sends a data group with errors, e.g. because a sensor-user is malfunctioning or data entered by a human has errors, it is usually the task of the functional process to determine if the event really occurred and/or if the entered data are really valid, and how to respond.

NOTE 4: A functional process is 'unique' (as in a) above), and its total size must be included in the size of the FUR, if it is initiated by a triggering Entry that results originally from a triggering event that is distinguished as unique within the FUR. Two or more functional processes within the same FUR may be unique even though they share some common functionality. See section 3.2.7 for examples of functional processes with shared functionality.

Functional process level of granularity. A level of granularity of the description of a piece of software at which

- the functional user (-types) are individual humans or engineered devices or pieces of software (and not any groups of these) AND

- single event (-types) occur that the piece of software must respond to (and not any level of granularity at which groups of events are defined). See Note 3 below.

NOTE 1: In practice, software documentation often describes functional requirements of different parts of the software at different levels of granularity, especially when the documentation is still evolving. Functional processes may be revealed at any of these levels of granularity.

NOTE 2: 'Groups of these' (functional users) might, for example, be a 'department' whose members handle many types of functional processes, or a 'control panel' that has many types of instruments, or 'central systems'.

NOTE 3: 'Groups of events' might, for example, be indicated in a statement of functional requirements at a high level of granularity by an input stream to an accounting software

system labelled 'sales transactions' or by an <u>input</u> stream to an avionics software system labelled 'pilot commands'.

Functional Size. A size of the software derived by quantifying the <u>Functional User Requirements</u>.[17]

Functional Size Measurement (FSM). The process of measuring <u>functional size</u>. [17]

Functional Size Measurement Method. A specific implementation of FSM defined by a set of rules, which conforms to the mandatory features of ISO/IEC 14143-1:1998. [17]

Functional User. A (type of) <u>user that is</u> identified in the Functional User Requirements of a piece of software being measured as a sender and/or an intended recipient of data processed by the software.

Functional User Requirements (FUR). A sub-set of the user requirements. Requirements that describe what the <u>software</u> shall do, in terms of tasks and services.

NOTE 1: Functional User Requirements relate to but are not limited to:

- data transfer (for example Input customer data, Send control signal);
- data transformation (for example Calculate bank interest, Derive average temperature);
- data storage (for example Store customer order, Record ambient temperature over time);
- data retrieval (for example List current employees, Retrieve aircraft position).

Examples of user requirements that are not <u>Functional User Requirements</u> include but are not limited to (although some of these may become true FUR by the time the solution is fully defined):

- quality constraints (for example usability, reliability, efficiency and portability);
- organizational constraints (for example locations for operation, target hardware and compliance to standards);
- environmental constraints (for example interoperability, security, privacy and safety);
- implementation constraints (for example development language, delivery schedule).

NOTE 2: In COSMIC documents, the term 'FUR' is restricted to mean the functional user requirements that:

- are derived from the available software artefacts (requirements, designs, physical artefacts, etc.);
- are adjusted, if necessary, by assumptions to overcome uncertainties in the available artefacts;
- contain all the information needed for a COSMIC Functional Size Measurement.

Otherwise we use expressions such as functional requirements', 'actual requirements' or 'physical artefacts', etc., depending on the context.

Input. The data moved by all the <u>Entries</u> of a given <u>functional process</u>.

Layer. A functional partition of a <u>software system</u> architecture

Level of decomposition. Any level resulting from dividing a <u>piece of software</u> into <u>components</u> (named 'Level 1', for example), then from dividing <u>components</u> into sub-<u>components</u> ('Level 2'), then from dividing sub-<u>components</u> into sub-sub components (Level 3'), etc.

NOTE 1: Not to be confused with '<u>level of granularity' which concerns</u> the level of detail of requirements.

NOTE 2: Size measurements of the components of a piece of software are only directly comparable for components at the same level of decomposition.

NOTE 3: Different levels of decomposition of a piece of software may correspond to different 'views' of the software's layers, e.g. as in Figure 2.4. However, software may be decomposed into 'levels' regardless of whether or not it is designed using a layered-architecture model.

Level of granularity. Any level of expansion of the description of any part of a single piece of software (e.g. a statement of its requirements, or a description of the structure of the piece of software) such that at each increased level of expansion, the description of the functionality of the piece of software is at an increased and uniform level of detail.

NOTE: Measurers should be aware that when requirements are evolving early in the life of a software project, at any moment different parts of the required software functionality will typically have been documented at different levels of granularity.

Measurement method [13]. A logical sequence of operations, described generically, used in the performance of measurements.

Measurement (Strategy) Pattern. A standard template that may be applied when measuring a piece of software from a given software functional domain, that defines the types of functional user that may interact with the software, the level of decomposition of the software and the types of data movements that the software may handle.

Model [15]. A description or analogy used to help visualize a concept that cannot be directly observed.

Modification (of the functionality of a data movement)

a) A data movement is considered to be functionally modified if at least one of the following applies:

- the data group moved is modified

- the associated data manipulation is modified

b) A data group is modified if at least one of the following applies:

- one or more new attributes are added to the data group

- one or more existing attributes are removed from the data group

- one or more existing attributes are modified, e.g. in meaning or format (but not in their values)

c) A data manipulation is modified if it is functionally changed in any way.

Non-Functional Requirement. Any requirement for the software part of a hardware/software system or software product, including how it should be developed and maintained, and how it should perform in operation, except a functional user requirement for software. Non-functional requirements concern:

- the software quality;

- the environment in which the software must be implemented and which it must serve;

- the processes and technology to be used to develop and maintain the software;

- the technology to be used for the software execution.

NOTE: System or software requirements that are initially expressed as non-functional often evolve as a project progresses wholly or partly into FUR for software.

Object of interest type. Any 'thing' in the world of the functional user that is identified in the Functional User Requirements about which the software is required to move a data group

in or out of the software, or to or from persistent storage. It may be any physical thing, as well as any conceptual object or part of a conceptual object.

NOTE 1: In the COSMIC method, the term 'object of interest' is used in order to avoid terms related to specific software engineering methods. The term does not imply 'objects' in the sense used in Object Oriented methods.

NOTE 2: When a functional user sends a data group describing itself, e.g. its state or its identity, or when a functional user receives data describing itself, then the functional user also fulfils the role of the 'thing' in its world, so it is also the object of interest described by the data group

NOTE 3: There is nothing absolute about an object of interest. A 'thing' may be an object 'of interest' to a functional user via one or more functional processes, but not be an object 'of interest' to another functional user via other functional processes, even in the same software being measured.

Operating environment (software).   The set of software operating concurrently on a specified computer system on which application software depends.

Output. The data moved by all the Exits of a given functional process.

Peer pieces of software. Two pieces of software are peers of each other if they reside in the same layer.

Persistent storage. Storage which enables a functional process to store data beyond the life of the functional process and/or which enables a functional process to retrieve data stored by another functional process, or stored by an earlier occurrence of the same functional process or stored by some other process.

NOTE 1: In the COSMIC model, persistent storage is a concept that exists only within the boundary of the software being measured, it cannot therefore be considered as a functional user of the software being measured.

NOTE 2: An example of 'some other process' would be in the manufacture of read-only memory.

Piece of software. Any discrete item of software at any level of decomposition from the level of a whole software system down to and including the level of the smallest component of a software system.

Purpose of a measurement. A statement that defines why a measurement is being made, and what the result will be used for.

R. Abbreviation for 'Read type'.

Read type. A data movement that moves a data group from persistent storage into the functional process which requires it.

NOTE: A Read type is considered to account for certain associated data manipulation – see the Measurement Manual for details.

Scaling (of a measurement). The process of converting a size measurement at one unit of measurement to a measurement at another unit of measurement.

Scope (of a measurement). The set of Functional User Requirements to be included in a specific functional size measurement instance. [17]

NOTE: (Specific to the COSMIC method.) A distinction should be made between the 'overall scope', i.e. all the software that should be measured according to the purpose, and the 'scope' of any individual piece of software within the overall scope, whose size should be measured separately. In the Measurement Manual, the term 'scope' (or the expression

'measurement scope') will relate to an individual piece of software whose size must be measured separately.

Software [17]. A set of computer instructions, data, procedures and maybe documentation operating as a whole, to fulfill a specific set of purposes, all of which can be described from a functional perspective through a finite set of Functional User Requirements, technical and quality requirements.

Software System. A system that consists only of software.

***Sub-process type**. A part of a functional process that either moves data (into the software from a functional user or out of the software to a functional user, or to or from persistent storage) or that manipulates data.

System. A combination of hardware, software and manual procedures organized to achieve stated purposes.

NOTE: The above definition is an adaptation of the ISO/IEC 15288:2008 definition. In the COSMIC definition, 'hardware, software and manual procedures' replaces 'interacting elements' in the ISO/IEC definition.

***Triggering Entry type**. The Entry data movement of a functional process that moves a data group, generated by a functional user, that the functional process needs to start processing.

NOTE: This definition results from the Generic Software Model, which is a logical model. Physically, a functional process may start processing before data has been entered e.g. when a human user clicks on a menu to display a blank screen for data entry.

***Triggering event type**. An event, recognized in the Functional User Requirements of the software being measured, that causes one or more functional users of this software to each generate one or more data groups. The first data group generated by any one functional user will subsequently be moved by a triggering Entry. A triggering event cannot be sub-divided and has either happened or not happened.

NOTE: Clock and timing events can be triggering events.

Unit of measurement [14]. A particular quantity, defined and adopted by convention, with which other quantities of the same kind are compared in order to express their magnitudes relative to that quantity. It is to be noted that units of measurement have conventionally assigned names and symbols.

See also 'COSMIC unit of measurement'

User [17]. Any person or thing that communicates or interacts with the software at any time.

NOTE: Examples of 'thing' include, but are not limited to, software applications, animals, sensors, or other hardware.

Value (of a quantity) [14]. The magnitude of a particular quantity, generally expressed as a unit of measurement multiplied by a number.

***W**. Abbreviation for 'Write type'.

***Write type**. A data movement that moves a data group from inside a functional process to persistent storage.

Note: a Write is considered to account for certain associated data manipulation – see the Measurement Manual for details.

***X**. Abbreviation for 'Exit type'.

APPENDIX G - CHANGE REQUEST AND COMMENT PROCEDURE

The COSMIC Measurement Practices Committee (MPC) is very eager to receive feedback, comments and, if needed, Change Requests for this guideline. This appendix sets out how to communicate with the COSMIC MPC. All communications to the COSMIC MPC should be sent by e-mail to the following address:

mpc-chair@cosmic-sizing.org

Informal general feedback and comments

Informal comments and/or feedback concerning the guideline, such as any difficulties of understanding or applying the COSMIC method, suggestions for general improvement, etc should be sent by e-mail to the above address. Messages will be logged and will generally be acknowledged within two weeks of receipt. The MPC cannot guarantee to action such general comments.

Formal change requests

Where the reader of the guideline believes there is a defect in the text, a need for clarification, or that some text needs enhancing, a formal Change Request ('CR') may be submitted. Formal CRs will be logged and acknowledged within two weeks of receipt. Each CR will then be allocated a serial number and it will be circulated to members of the COSMIC MPC, a world-wide group of experts in the COSMIC method. Their normal review cycle takes a minimum of one month and may take longer if the CR proves difficult to resolve. The outcome of the review may be that the CR will be accepted, or rejected, or 'held pending further discussion' (in the latter case, for example if there is a dependency on another CR), and the outcome will be communicated back to the Submitter as soon as practicable.

A formal CR will be accepted only if it is documented with all the following information.

- Name, position and organization of the person submitting the CR.
- Contact details for the person submitting the CR.
- Date of submission.
- General statement of the purpose of the CR (e.g. 'need to improve text…').
- Actual text that needs changing, replacing or deleting (or clear reference thereto).
- Proposed additional or replacement text.
- Full explanation of why the change is necessary.

A form for submitting a CR is available from the Knowledge base of www.cosmic-sizing.org . The decision of the COSMIC MPC on the outcome of a CR review and, if accepted, on which version the CR will be applied to, is final.

Questions on the application of the COSMIC method

You can use the forum on cosmic-sizing.org/forums to post your questions and receive answers from our world-wide community. The quality of any answers will depend on the knowledge and experience of the community member that writes the answer; the MPC cannot guarantee the correctness. Commercial organizations exist that can provide training and consultancy or tool support for the method. Please consult the www.cosmic-sizing.org web-site for further detail.