The COSMIC Functional Size Measurement Method
Version 4.0.2

Guideline for Sizing Real-time Software

VERSION 2.0
November 2019

Foreword

Purpose of this Guideline and relationship to the COSMIC Measurement Manual

To help those working in the domain of real-time software to map the concepts they typically use to determine and model the requirements of real-time software, to the concepts of the COSMIC method of measuring a functional size of software. The Guideline also provides many measurement examples and illustrative cases.

- The Guideline is hence an aid-to-translation from the terminology used by real-time software practitioners to the terminology of the COSMIC method. No additional principle or rule is required to apply the COSMIC method to the real-time domain, beyond those that are provided in the COSMIC Measurement Manual [1].

Intended readership of the Guideline

To be used by anyone involved in defining, specifying, developing and managing software products in the real-time domain [2]. This includes the members of the software metrics team and/or developers who have the task of measuring the functional size of real-time software according to the COSMIC method. It should also be of interest to those who have to interpret and use the results of such measurements. This Guideline is not tied to any particular real-time software development methodology or life-cycle model though some examples refer to specific real-time requirements determination or modeling methods. Note that COSMIC does not endorse any particular method or tool.

Readers of this Guideline are assumed to be familiar with the COSMIC Measurement Manual [1]. For ease of maintenance, there is little duplication of material between that document and this Guideline.

Scope of applicability of this Guideline

This Guideline concerns the measurement of 'real-time' software, where we use this term in a broad sense. According to Wikipedia, 'a system is said to be *real-time* if the total correctness of an operation depends not only upon its logical correctness, but also upon the time in which it is performed. Real-time systems, as well as their deadlines, are classified as 'hard', 'firm' or 'soft' depending on the consequence of missing a deadline.'

For the purpose of this Guideline: it includes any software whose operation is controlled by a clock or timer mechanism. The COSMIC method can be used to measure the functionality of all these various types of 'real-time' software. (However, it should be noted that a specific timing constraint, or 'deadline' such as 'all commands must be satisfied within 1 millisecond' is a non-functional requirement. COSMIC functional sizing can measure any functionality needed to achieve this constraint but the specific numerical value of the constraint (1 millisecond, or 1 microsecond, or whatever) does not actually affect the software functional size.)

Examples of real-time software include the monitoring and control of industrial systems, automated acquisition of data from the environment and from scientific experiments, the monitoring and control of vehicle systems such as engines, ventilation, collision-avoidance, etc. and of household appliances. On the large scale, real-time systems control the world's telephone networks, individual aircraft and air traffic, power plants and such-like. Some software systems such as hotel or airline reservation systems may be described as hybrids of business application software and real-time software, because they must process enquiries and bookings within real-time constraints. Finally, middleware and infrastructure software such as operating systems provide basic tasks and services for real-time applications and hence operate within real-time constraints.

Introduction to the contents of the Guideline

Chapter 1 discusses the characteristics of real-time software systems, the way their requirements are stated and how they can be mapped to COSMIC method concepts. Chapter

2 deals with the measurement strategy and in particular with identifying the functional users of the software to be measured. Chapter 3 discusses the mapping and measurement phases. Chapter 4 presents a number of illustrative examples.

For definitions of the terms of the COSMIC method in general, please refer to the Glossary in the Measurement Manual [1]. Terms specific to the real-time software domain are defined in a separate Glossary at the end of this Guideline. Note that the literature on information technology uses a number of terms that are used with various meanings, but which are defined in the COSMIC method with very specific meanings. The Measurer must therefore be careful to correctly apply the terminology of the COSMIC method when using this Guideline.

This version 2.0 of this Guideline complies with v4.0.2 of the COSMIC method.

For a list of the significant changes that have been made from the previous version, see the Guideline Change History in the Appendix.

The COSMIC Measurement Practices Committee.

Table of Contents

1

MAPPING REQUIREMENTS OF REAL-TIME SYSTEMS SOFTWARE TO COSMIC CONCEPTS

The purpose of this chapter is to relate the terminology and concepts from the domain of real-time software, as used in various methods of expressing real-time system requirements, to the concepts of the COSMIC functional size measurement method. If a functional size must be measured from the artefacts of some existing operational software, the Measurer should be able to use this same mapping of concepts to reverse engineer from the artefacts to the original functional user requirements (FUR), expressed in COSMIC terms, which can then be sized.

Recalling the key features of the COSMIC method, a measurement should proceed in three phases, see [1].

In the *Measurement Strategy* phase, the aim is to determine the purpose of the measurement, hence the scope of the software to be measured and its functional users, i.e. the people or 'things' that are the senders or intended recipients of data to/from the software to be measured.

The level of granularity of the requirements and the level of decomposition of the software to be measured are also determined in this phase.

In the *Mapping* phase, the aim is to map the FUR of the software to the concepts of the COSMIC method. We use 'FUR' to mean only the functional user requirements that are completely defined so that a precise COSMIC functional size measurement is possible. For requirements in general or where requirements have been specified at a 'higher' level of granularity, i.e. they have not yet evolved to a level of detail where a precise COSMIC size measurement is possible, we will use 'requirements' or 'functional requirements', as appropriate.

The *Mapping* phase has two main steps.

- Identify the 'triggering events' detected by (or generated by) the functional users that the software must respond to, and hence the corresponding 'functional processes' (see section 3.1)
- Identify the 'objects of interest' and 'data groups' referenced by the piece of software to be measured and hence the 'data movements' (Entries, Exits, Reads and Writes) in each functional process (see section 3.2)

In the *Measurement* phase (see section 3.3), the functional size of a piece of software is measured by counting the total number of data movements summed over all its functional processes. The functional size of a change to the requirements is measured by counting the total number of data movements that must be added, modified or deleted to satisfy the change requirement.

Note that whenever we mention 'triggering events', 'functional processes', 'data movements', etc., we mean 'types' of these, *not* 'occurrences' (see the Measurement Manual [1], section 1.3.3).

1.1 Characteristics of real-time systems software

Some of the characteristics of real-time systems relevant to measuring the functional size of the software are treated in the following sections.

1.1.1 Event-driven systems

Real-time software is often characterized as 'event-driven', i.e. its functionality must respond to events with real-time constraints. Its behaviour can be illustrated as a finite state machine where each event (occurring outside the software) that the software must respond to may affect the state(s) of the software. The state does not necessarily change; for instance an enquiry functional process leaves the state of the machine unchanged on completion.

A key concept of the COSMIC method is that an event is sensed by, or in real-time software sometimes generated by, a functional user of the software being measured. A functional user communicates the occurrence of an event to the software by sending a data group that is moved into a functional process by an Entry data movement, which triggers the functional process to start executing. This is shown in the following diagram taken from the COSMIC Measurement Manual [1].

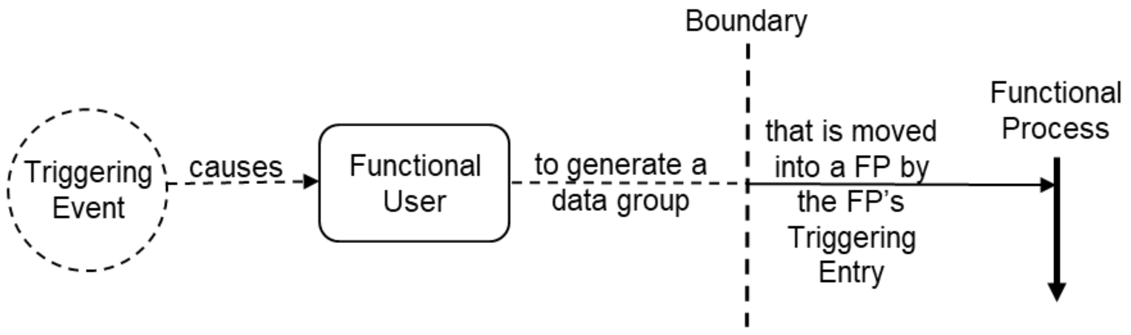

Figure 1.1 – Relation between triggering event, functional user and functional process

Three examples of triggering events help to explain the relationships shown in Figure 1.1:

EXAMPLE 1: A sensor detects a stimulus to which the software must respond.

- *the triggering event is the stimulus that the sensor is designed to detect;*
- *the functional user is the sensor;*
- *the sensor generates and sends a message (a data group), which is moved into a functional process by its triggering Entry data movement, informing that the event has occurred; this message may also carry other data about the triggering event.*

EXAMPLE 2: A piece of software A must pass a request to a piece of software B for a service.

- *software A effectively generates the triggering event when it needs the service from software B by generating the request for service (a data group that provides the input data needed for the service);*
- *software A is the functional user of software B;*
- *the request for service message is moved into a functional process in software B by its triggering Entry; the functional process can then provide the service.*

EXAMPLE 3: A piece of software must execute a control process each time a clock 'ticks'.

- *the clock effectively generates the triggering event by generating a 'tick' (a data group);*
- *the clock is a functional user of the software;*
- *the 'tick data group' is moved into a functional process by its triggering Entry to start its task.*

In all three examples the 'task' (the service) that the software must undertake, to respond to a triggering event, is a 'functional process', which is a sequence of data movements that is not complete until it has done all that is needed to meet its FUR for all possible responses to its triggering Entry.

Note some important points:

- Each functional process is independent of any other functional process, However, there can be various cardinalities (1:n, 1:1, or n:1) along the chain of triggering event – functional user – data group - functional process of Figure 1.1 (see the Measurement Manual [1] for examples).

- It may be that a series of functional processes can only occur in a particular sequence but this does not affect the fact that each functional process must be separately triggered. Example: a ‘stop process’ cannot occur before a ‘start process,’ but each process is triggered by a separate event.
- A measured functional size does not take into account:
- a) any specific real-time timing constraint, e.g. that a response is required in less than one millisecond, as this is a non-functional requirement (however, see section 4.2 for the measurement of timer functionality);
- b) that some functional processes may be triggered by ‘routine’ triggering events, when the software is waiting for the triggering Entry, and others by ‘exceptional’ triggering events that give rise to interrupts (see next section 1.1.2).

1.1.2 Interrupts

An Interrupt Message is generated when a Functional User detects an event that requires the current activity of a system to be changed[1]. If a functional process has been triggered and started executing but has not yet terminated normally, an Interrupt Message may have one of two possible consequences.

- If interrupts are handled by an interrupt handler outside the scope of the software being measured, e.g. in another layer than that of the software being measured, then by definition the functional processes of the software being measured do not need to take any account of the fact that interrupts may occur.
- Alternatively the interrupt may be passed to the executing functional process as an additional Entry to its normal triggering Entry. The action the functional process must take on receipt of the Interrupt Entry will depend on its FUR. The action may result in additional data movements to those needed for normal execution and should be measured according to the normal rules.

1.1.3 Functional size may vary with the functional users of real-time software

The most common purpose of measuring a functional size of real-time software is for project performance measurement and/or project effort estimation. For most measurement purposes and scopes, the functional users of real-time software will be identified as the ‘things’ that interact *directly* with the software being measured. Typically these may be:

- hardware input devices (e.g. sensors, measurement devices, a clock or timer);
- hardware output devices (e.g. an actuator, a display, a communications line);
- other pieces of software or hardware that may send input and/or receive output.

However, the measurement purpose might be to size the functionality as seen by a human operator of a real-time system (e.g. the functionality provided at the operator workstation of a process control system or by the operator interface for a simple copier or mobile telephone).

Humans interact *indirectly* with software. They are only aware of the functionality that is made available to them via their input-output interface. This functionality may be much less than the total functionality that the software must provide. It is therefore very important when measuring the functional size of real-time software to define clearly the functional users for which the measurement is valid.

For more on types of functional users, see section 2.2.

1 The ISO/IEC/IEEE 24765:2010 Systems and software engineering—Vocabulary standard defines ‘interrupt’ as ‘(1) the suspension (or termination) of a process to handle an event external to the process (2) to cause the suspension (or termination) of a process (3) loosely, an interrupt request.’ (The words in brackets ‘or termination’ have been added for this Guideline.)

1.1.4 Embedded, executing on an operating system or federated?

We use the term ‘application software’ for any software developed by order of a stakeholder to perform a particular set of tasks, and the term ‘infrastructure software’ for operating systems, software data handlers and device drivers that support applications.

Real-time application software may be embedded on a chip (System on Chip), such as a field-programmable gate array (FPGA) or a programmable logic controller (PLC), which may itself be specialized for the application, e.g. to survive rugged operating conditions. Very commonly, a simple embedded software application interacts directly with various input/output hardware devices, so does not need an operating system.

Alternatively, real-time application software may be installed on a general purpose or specialized processor and execute with the support of a real-time operating system (RTOS). The RTOS may handle all messages to and from the various hardware input/output devices; it will reside in a different layer (an infrastructure or ‘lower’ layer) of the software architecture from the application. If the purpose is to measure the application software, the presence of the RTOS and any other layers of infrastructure software must be ignored, since a principle of the COSMIC method is that the scope of a measurement must be confined to one layer. The same principle applies to a need to measure some software in any of the infrastructure software layers; the scope of a measurement must always be confined to software within one layer.

Federated systems, or a ‘system of systems’ comprise multiple elements of the above types of processors that communicate over a common bus or over a network. Each element in the system should be measured separately.

1.2 Statements of requirements

1.2.1 The problem of allocation of requirements to hardware or software

A difficulty for measuring a functional size of real-time software is that often the requirements are stated at the ‘system’ level, i.e. before they are allocated to hardware or software (e.g. in the System on Chip concept), rather than explicitly at the software level. Obviously, until a decision has been made - and documented - on which requirements will be allocated to software, it is very difficult to agree on measurement results because various Measurers may make different assumptions on what the software will do.

In principle the COSMIC method can be applied to functional requirements for information processing 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 COSMIC without any knowledge of what hardware or software (if any) is involved.

If system requirements must be measured for which their allocation between software and hardware is not clear and no expert advice is available to decide on the allocation, the Measurer should document any assumptions about the allocations and measure the software part. The lower and upper limits of the expected software functional size should also be indicated.

1.2.2 Non-functional requirements

For non-functional requirements, please see the Measurement Manual [1] and the ‘Guideline on Non-Functional & Project Requirements’ [5].

2

THE MEASUREMENT STRATEGY PHASE

Determining the ‘strategy’ for a COSMIC functional size measurement requires that various parameters be considered before starting an actual measurement. The parameters, to be discussed in this chapter, are:

- the purpose and scope(s) of the measurement;
- The *layer(s)* of the software architecture, i.e. the layer in which each piece of the software to be measured resides
- the functional users of the software to be measured (the ‘things’ that are the sources and intended recipients of the data to/from the software to be measured;
- the level of granularity of the functional requirements of the software to be measured;
- the level of decomposition of the software itself.

The effort needed to determine the strategy is usually trivial but recording these parameters helps ensure that the resulting size measurement can always be interpreted reliably, i.e. future users of the measurement can always be sure they are comparing ‘apples with apples’.

As an aid to determining a measurement strategy, the Guideline for 'Measurement Strategy Patterns' [4] describes, for each of several different types of software, a standard set of parameters for measuring software sizes, called a ‘measurement strategy pattern’.

2.1 The purpose and scope of the measurement

2.1.1 The measurement purpose

The purpose of a measurement defines why a measurement is required, and what the result will be used for.

2.1.2 The measurement scope

The purpose of the measurement determines the scope of the measurement (which defines the extent of the functionality to be measured). A particular purpose may require more than one piece of software to be measured separately, e.g. for a project that must deliver multiple pieces of software, there would be more than one measurement scope.

The scope of a piece of software to be measured must be confined to a single software layer. For more on distinguishing layers, see the Measurement Manual [1]

2.2 Identifying the functional users

When measuring real-time software, the functional users (‘the senders and/or intended recipients of data’) that interact with the software being measured will be typically any of the following:

- a clock or timer (See the Glossary for use of these terms in this Guideline);
- sensors (e.g. of temperature, pressure, voltage) that provide input, either when polled, or via interrupts, or by sending their data and/or status at intervals;
- hardware devices that receive output (e.g. a valve or motor actuator, switch, lamp, heater);
- hardware chips, having the ability to trigger functional processes (e.g. watchdog chips);
- ‘dumb’ hardware memory such as a ROM which can only respond to a request for data;
- communications devices (e.g. telephone lines, computer ports, aerials, loudspeakers, microphones);

- hardware devices with which humans interact (e.g. push buttons, keyboards or displays);
- other pieces of software that supply data to or require data from the software being measured.

It is valuable to draw a ‘context diagram' that shows the interaction of the software being measured with its functional users and with persistent storage, if used. When drawing a context diagram, it may be helpful to distinguish functional users that are:

- sources of triggering events (and therefore of data groups that are moved by triggering Entries)
- sources of other input data (e.g. that may be polled to provide data groups for non-triggering Entries)
- intended recipients, or destinations of data (to which Exits are sent)

Some functional users may fulfil more than one of these roles, e.g. other pieces of software that interact with the software being measured, intelligent hardware devices or communication lines.

All the above types of functional users may interact with the software being measured either directly, or indirectly, e.g. via an operating system or simple device driver software. However, the functionality of this ‘enabling’ software should be ignored (unless, of course, it is the subject of the measurement).

Real-time software may also be measured from the viewpoint of humans as functional users (‘the senders and/or intended recipients of data’) that interact indirectly with the software being measured e.g. operators that start and stop the system, set parameters, monitor displays of operational performance, need to be notified of emergency conditions, etc.

Consider the case where a button is pushed by a human operator. Is the functional user taken to be the button that interacts directly with the software to be measured, or the human that presses the button that interacts indirectly with the software? (They ’see’ different events. For the button, the event is ‘I have been pressed’. For the human operator, the event is perhaps an emergency condition that means he/she must raise an alarm). The choice depends on the purpose of the measurement. This choice of functional users must be one or the other; it makes no sense to measure a size, or to sum two sizes, as seen by a mix of functional user types (humans, hardware devices or other pieces of software). The following Examples illustrate where there is a choice of functional user.

EXAMPLE 1. Section 4.1.1 describes an industry process which is controlled by a programmable logic controller (PLC). The purpose is to measure the size of all the embedded software functionality needed to make the system work, not just the limited view of the functionality as seen by a human operator. The process is started by a human operator pushing a start button. But in this example, given the measurement purpose, the start button is considered to be a functional user, not the operator who pushes it to start the process.

EXAMPLE 2. The embedded software of a mobile phone (‘cellphone’) has to interact with several types of buttons, a screen (which may serve as an input device as well as output display), its battery, loudspeaker, aerial, etc. A human user of such a phone sees only a small part of the functionality that the software needs to provide its services. So it is possible to measure two functional sizes, depending on the choice of functional users.

Toivonen [3] measured the functionality as seen by human users of two mobile phones in order to compare their ‘packing density’ (functional size / memory size). This was an important economic measure for the phone manufacturer. But the software sizes measured by Toivonen were much smaller than the sizes that the software engineers would have to develop to provide all of the functionality necessary for the phone to meet all the requirements of all its hardware/software functional users.

Determining the functional users depends on the requirements, as the following example shows.

EXAMPLE 3. One or more buttons (-types)?

Consider a factory that has a moving production line that can be stopped by pushing a button; there are buttons at several different locations along the line. Should the Measurer identify one or several functional users (types)? The answer depends on the functional 'user' requirements that must be measured. The issue from this example is whether pressing the buttons leads to different triggering events and separate functional processes, e.g.

a) Requirements: Any operator may press a button to stop the line in an emergency. When a button is pressed, the system logs the time at which the line was stopped and the button that was pressed. There are many buttons along the line that all have the same effect. As the buttons are subject to the same FUR ('pressing any button must stop the line'), identify only one functional user type and one functional process type to meet these FUR;

b) Requirements as case a) but there is also a requirement for a button in a supervisor's office which is used by the supervisor to stop the line at the end of the work-day. If it has only the same effect as case a), then still identify only one functional user type – again they are subject to the same FUR - and one functional process type.

c) Requirements as case b) but in addition to its use for stopping the line at any time, there is a requirement that when the button in the supervisor's office is pressed AND held down for three seconds, the system stops the line and then produces a log of the day's stop/start events. (The timing of the three seconds is controlled by the button itself.) We now have two functional users (any stop button on the line, and the supervisor's stop button) and two functional processes. The two functional processes share some functionality (stopping the line) but are invoked by different triggering events (emergency stop, and end-of day stop) and have different effects.

d) Requirements as case c) but there is an additional requirement that the supervisor has a second button that when pressed will start or re-start the line after it had been stopped and log the start time. Now we have three functional user types (any stop button on the line, and the supervisor's stop and start buttons) and 3 functional processes (stop the line, stop the line and produce a report from the supervisor's first button, and start or re-start the line from the supervisor's second button).

See also section 2.3 and real-time example 1 in section 3.5.9, both of the Measurement Manual [1]. See also the Tire Pressure Monitoring System case in section 4.5 of this Guideline.

A context diagram should show persistent storage if any of the software's functional processes are required to store data persistently (that is beyond the life of the process), or to retrieve data that has been stored persistently by another process. In the COSMIC Generic Software Model, persistent storage is available to any software being measured. The term does not imply any type of physical storage.

2.3 Identifying the level of decomposition and the level of granularity

Before starting a measurement of requirements, two aspects of the requirement artifacts must be considered to ensure that the measurement will satisfy its purpose, be as accurate as needed, and be interpreted with certainty by future users:

- The 'level of decomposition' of the software. This refers to the sub-division of the software within its layer into separate components that may exchange or share data. The process of sub-division may start during the requirements definition stage when an initial requirement is seen to be too large to be handled by one team and so it is decided to sub-divide the system into different sub-systems, sub-sub-systems, etc., which may be implemented at different times. Alternatively, sub-division into different components may arise if the software must be distributed over different processors.
- The 'level of granularity' of the requirements for the software and/or its components, which concerns their level of detail. Often, in the life of a project, requirements are produced in a

'top-down' way, i.e. first in outline form, then as the project progresses being worked out in more and more detail (in other words at lower and lower levels of granularity).

A precise COSMIC size measurement is possible only when the detail is sufficient to identify all the individual events that the software must respond to and hence the functional processes and their data movements. If a size measurement is required before these details are available, then an approximation variant of the COSMIC method may be used. These variants involve scaling sizes measured at the actual level of granularity of the requirements to the level of the functional processes and their data movements. Care must be taken with these methods since, at a given point in time, different parts of the requirements may have been worked out at different levels of detail.

Note that any decomposition of the software should be determined *before* the level of granularity of the requirements, as the latter might vary from one component to another. As requirements and the software design evolve, both parameters should be monitored for their effect on the measurement approach.

There is nothing specific to real-time software in considering these two factors. More detail on these two factors is given in the Measurement Manual [1]. The 'Guideline for early or rapid COSMIC sizing of functional requirements' [6] discusses several variants for approximate sizing. It includes a worked example of sizing the requirements of a telecoms software system at successively lower levels of granularity,

3

THE MAPPING AND MEASUREMENT PHASES

3.1 Identifying the triggering events and functional processes

As described in section 1.1.1, software is triggered to do something by ‘events’ in the world of its functional users. Identifying the triggering events is therefore of critical importance because it enables the Measurer to identify the 'somethings', namely the functional processes. The steps for identifying functional processes in the functional user requirements (FUR) of a piece of software are as follows:

1. 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 from state diagrams and in entity life-cycle diagrams, since state transitions and entity life-cycle transitions in the world of the functional users that the software must react to correspond to triggering events);
2. Identify which functional user(s) of the software may respond to each triggering event;
3. Identify the triggering Entry (or triggering Entries, one for each functional process to be triggered) that each functional user may initiate in response to the event (there may be non-triggering Entries as well);
4. Identify the functional processes, each started by its triggering Entry.

EXAMPLE: The speedometer software of a car is connected to a rotation measurement sensor located on the drive shaft that measures its revolutions per minute (rpm), and to a key-in sensor, a clock, and a display unit for the driver. The software's persistent storage contains the parameters needed to send messages to a pre-defined variety of display units. The speedometer software is required to capture at key-in time the display parameters and initialize the installed display unit. A clock triggers the software at five millisecond intervals to capture rpm information from the drive shaft, calculate the speed, and send the speed to update the display unit using parameters appropriate for this display unit.

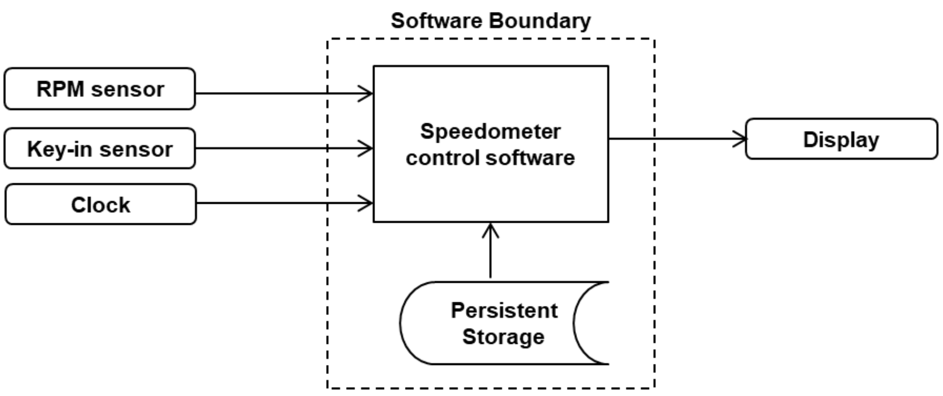

Figure 3.1 – Context diagram for the speedometer software

The context diagram shows the four functional users of the speedometer software, namely three input devices (the rpm sensor, the key-in sensor and the clock) and the one output device (the driver display).

There are two events that need to be responded to by the speedometer software (i.e. are triggering events), They are the key-in event and the 5 millisecond clock tick. Hence the speedometer control software has two functional processes, FP1 and FP2.

- *FP1 initializes the speedometer control software on the event of 'key-in' detected by the key-in sensor, which includes reading the parameter data for the display;*
- *FP2 measures the speed on the event of the tick generated by the clock every 5 ms and sends the speed to the display.*

3.2 Identifying objects of interest, data groups and data movements

3.2.1 Objects of interest and data groups

Any functional process comprises sub-processes, called 'data movements' that both move data and are considered to account for related data manipulation. A data movement moves a data group whose attributes describe a single 'object of interest'.

In real-time software a data group often comprises one or a few data attributes, sent from an input device, e.g. a sensor to software, or of a signal sent from software to an output device, e.g. an actuator. The object of interest described by the data group can be determined from the devices involved. For instance, the speedometer control software in the example of section 3.1 has the 'RPM sensor' as a functional user. This sensor sends a data group to the software, which has one attribute 'current rpm'. The object of interest of this data group could be considered as the drive shaft or the rpm sensor. It is often the case in real-time software that a functional user (the RPM sensor in the example above) is also the object of interest of a data group that it sends (i.e. it is sending data about itself). For more on 'The functional user as object of interest' see section 3.3.5 of the Measurement Manual [1].

3.2.2 Data movements

There are four sub-types of data movements: Entry, Exit, Read, and Write.

Entry and Exit data movements move a data group across a software boundary, from or to a functional user respectively. Read and Write data movements move a data group from or to persistent storage respectively.

For real-time software, the rules of section 3.5.9 of the Measurement Manual [1] 'When a functional process requires data from a functional user' are particularly important. Figure 3.2 shows the various ways in which real-time software can receive or 'get' a data group from its functional users (which varies with their capabilities) and from persistent storage.

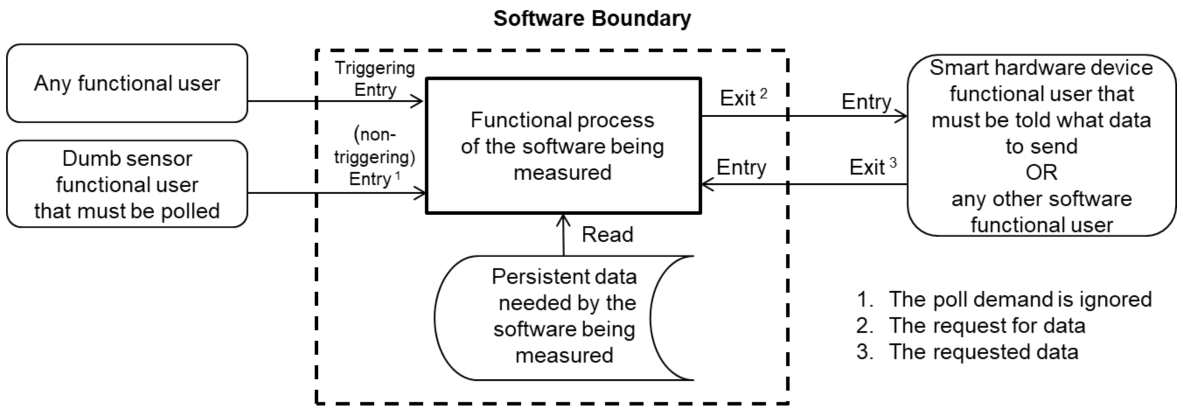
Software Boundary
Any functional user
Triggering Entry
Dumb sensor functional user that must be polled
(non-triggering) Entry 1
Functional process of the software being measured
Exit 2
Entry
Entry
Exit 3
Smart hardware device functional user that must be told what data to send OR any other software functional user
Read
Persistent data needed by the software being measured
1. The poll demand is ignored
2. The request for data
3. The requested data

Figure 3.2 - The various ways in which a functional process can receive or get data

A data group may be present in persistent storage and be available to be read either:

- by the data group having been made persistent by a Write data movement of an earlier occurrence of a software functional process, or
- by being stored in physical read-only memory during the manufacture of a chip in which software will be embedded. Parameter data needed by the embedded software may be stored this way.

It may be queried why a whole Entry data movement (worth one COSMIC Function Point, or CFP) is measured when the signal may be a single bit, as in the case of a clock tick. But remembering that each data movement is assumed to account for the associated data manipulation, a triggering Entry must account for initializing sub-processes, not just the movement of one bit.

As well as receiving or getting data groups, a functional process can, of course, also:

- send out data groups to a functional user via an Exit data movement where no response is expected,
- and can 'put' data groups to persistent storage via a Write data movement.

3.2.3 Data manipulation

The COSMIC method was not designed to account explicitly for data manipulation. As noted above, the method assumes that data manipulation is accounted for by each data movement.

However, the method may be reasonably used to measure certain types of software that require extensive data manipulation. See section 4.5.2 of the Measurement Manual [1]. 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 'reasonably used', we mean that the method has produced meaningful and useful sizes in relation to the purpose of the measurement, e.g. project performance measurement, estimating, benchmarking and such-like. 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. See section 4.7 of this Guideline for two examples of the measurement of software that includes significant data manipulation.

If the software that must be measured is mostly 'movement–rich' but includes some significant, localized mathematical algorithms, the COSMIC method allows for a 'local extension' whereby an organization can define its own local scale for sizing algorithms alongside the CFP scale for sizing software functionality. Alternatively, if the purpose is estimating project effort, the measurement scope can be defined to exclude the algorithms. The sizing and estimating process can then be applied only to the 'movement-rich' functionality whilst estimating for developing the algorithms can be dealt with by another appropriate process. For more on this subject see section 4.5.5 of the Measurement Manual [1], entitled 'Local extension with complex algorithms'.

3.2.4 Error or fault messages in real-time software

Messages containing error or fault indications in real-time software that are intended for hardware or software functional users (i.e. NOT for human functional users) must be analyzed in the same way as any other data movement. Two cases arise.

1. 'In-line' error message: If a data group describing a particular object of interest in a message issued by the software being measured may include an indication of a fault or of an error in place of the normal valid data, this fault/error indication describes the same object of interest as the normal valid data. Hence this data group is moved by only one Exit, i.e. the fault/error indication is <u>not</u> identified as a separate Exit.

   EXAMPLE: Output= {VAL1, VAL 2, VAL3, …, VAL n, ERROR} where 'VAL' indicates a valid value.

   Here, we don't really care about the type of the ERROR because nothing is implemented to handle this ERROR (e.g. LOG modules, alternative analysis modules, etc.)

2. Separate error message: If the software being measured issues the reason for a fault or error condition as a separate message, then a separate Exit may be identified. To be

measured as a separate Exit, the error message must describe a different object of interest than the message containing the normal valid data and/or the error message must be intended for a different functional user than the user that would receive the normal valid data. (See the 'Data Movement Uniqueness' rules in section 3.5.7 of the Measurement Manual [1]).

The general case is:

Output1= {*VAL1, VAL 2, VAL3, …, VAL n, ERROR*}

Output2= {Sensor_failure, Internal_error, General Failure*, …*}

EXAMPLE: Suppose a busy airport has multiple radar stations to control incoming air traffic on a runway. In a particular case, the radar station software is asked to report the number of aircraft in the 45-135 degree quadrant around runway 09L. Two of the radar stations report 2 aircraft and the third radar station reports 5. The software will report 2 aircraft to the traffic controllers as a majority response to the request, but will also report a separate warning message to the controllers and to the radar engineers that this is a majority decision and that one station reports 5 incoming aircraft.

The object of interest of the first message is 'Incoming air traffic on runway 09L'.The object of interest of the error message could be 'Radar station disagreement'

For definitions and other examples of error messages in real-time software, see the Measurement Manual [1], section 3.5.11, Real-time examples 1 and 2.

See also the cases in sections 4.1, 4.2, 4.3 and 4.5 of this Guideline, which all include messages indicating error or fault conditions that are measured as normal Exits.

3.3 Measurement and measurement reporting

See the Measurement Manual [1] for all principles and rules for:

- aggregating measurement results
- measuring the size of changes to software
- measurement reporting

All of these topics are domain-independent, i.e. there is nothing specific to real-time software.

4

EXAMPLES

4.1 Industry automation and the PLC

4.1.1 The programmable logic controller (PLC)

Programmable Logic Controllers (PLC's) are computers with extensive input and output facilities that can be connected to sensors, actuators and such-like. Many industrial processes are controlled by PLC's including conveyor belts and associated machinery, flat-product fabrication, assembly-line manufacturing and chemical processes.

4.1.2 PLC software for controlling a process in a chemical factory

Requirements

A process in a chemical factory is controlled by a PLC. The process consists of filling a tank with a liquid, heating the liquid and then emptying the tank when a temperature is reached that is pre-set in the temperature sensor device.

In the following description of the requirements of the process (system) control we assume that all mentioned functionalities are allocated to the PLC software, unless stated otherwise.

- The process is started by a human operator pressing a start button connected to the PLC which controls all the subsequent steps.
- The software issues a command to open the inlet valve of the tank and the tank fills with liquid under gravity.
- When the tank is full ('high level reached' is detected by the high level sensor) the software receives a message from this sensor and sends commands to close the inlet valve and to start the heater to heat the liquid.
- When the software is informed that the pre-set temperature is reached, it sends commands to stop the heater, open the outlet valve and start the pump to empty the tank.
- The pump continues emptying until 'low level reached' is detected by the low level sensor. On receipt of a message from this sensor, the software sends a command to stop the pump.
- During the entire the process, the process status ('Filling', 'Heating', 'Pumping') is shown on an operator display controlled by the software. When the process is finished, the software causes an audible alarm to sound and the message 'Process finished' is shown on the display.
- When the process is started and whilst the process is running, the PLC software polls the valves, the heater and the pump asking for their status at regular intervals to detect any fault conditions.
- If the PLC software is informed that an error is detected, it starts the audible alarm and displays a message to the operator showing the device(s) concerned. If an operator receives an error message, the operator deals with it manually, outside the software system.
- The polling frequency is determined by signals ('ticks') from a clock.

Context diagram

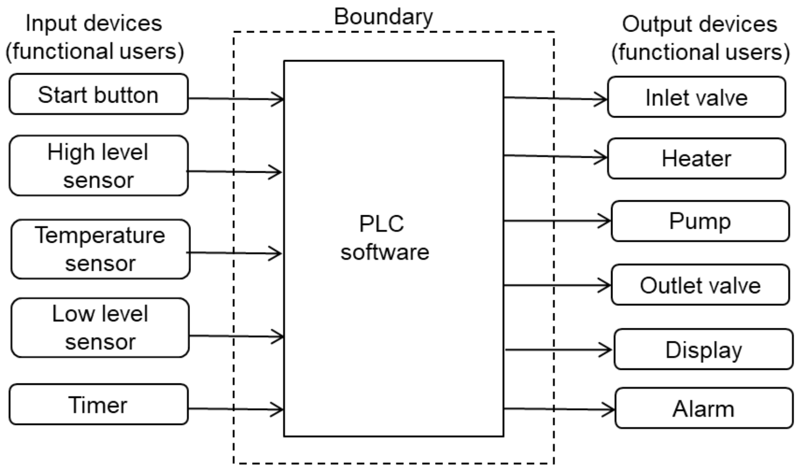

Figure 4.1 - Chemical Factory Process: PLC software Context Diagram

Analysis

The measurement assumes all the hardware devices that interact directly with the software are its functional users, as shown in the context diagram. The PLC does not have an operating system.

In the requirements as described above, the software is not decomposed in any way and is not a component of another piece of software. The level of granularity of the requirements is at the 'functional process level of granularity', i.e. the level of individual functional users and events (rather than groups of these).

There are several triggering events (events the software is required to respond to) and corresponding functional processes (that respond to the triggering events):

Table 4.1 - Chemical Factory, triggering events and functional processes

Triggering event	Functional user that initiates the functional	Corresponding Functional process
Start button pushed	Start button	Start process/Fill tank
High level reached	High level sensor	Heat liquid
Pre-set temperature reached	Temperature sensor	Stop heating/Empty tank
Low level reached	Low level sensor	Finish process
Clock tick (= time to poll)	Clock	Fault check

Each data group moved must describe an aspect of a single object of interest. The data groups consist of the signals from the start button, the sensors and the clock to the software and the signals from the software to the actuators, valves and the devices for the operator.

As noted in section 3.2.1, in this case the object of interest of each data group entering the software is the functional user that sent the group (i.e. the functional user is sending data about itself). Similarly the object of interest of each data group that leaves the software is the functional user that receives the group (i.e. the functional user is being sent data about itself). For instance, the data movements starting or stopping the pump move data groups that specify the (desired) states of the pump. The pump is therefore the object of interest of these data groups.

The software determines the process status to be displayed from the triggering Entry for each functional process (except the Fault Check process). For instance, from the start button signal the software determines that the current status is 'Filling' and displays this status.

The functional processes of the PLC software are as follows. The data movements (abbreviated as DM), the data groups moved and an explanation are shown for each functional process. We assume that the devices that the software polls to determine their status are ‘dumb’, i.e. the software inspects the state of these devices, which requires only one Entry per device type for the poll (see the Measurement Manual [1], section 3.5.9).

Functional process: Start process/Fill tank

DM	Functional User / Object of interest	Data Group
Entry	Start button	Start process message
Exit	Inlet valve	Open inlet valve command (to start entering
Exit	Clock	Start clock command (for fault detection at regular intervals)
Exit	Display	Display status command (‘Filling’)

The size of this functional process is 4 CFP.

Functional process: Heat liquid

DM	Functional User / Object of interest	Data Group
Entry	High level sensor	Tank full message
Exit	Inlet valve	Close inlet valve command (to stop liquid entering)
Exit	Heater	Start heating command
Exit	Display	Display status command (‘Heating’)

The size of this functional process is 4 CFP.

Functional process: Stop heating/Empty tank

DM	Functional User / Object of interest	Data Group
Entry	Temperature sensor	Pre-set temperature reached message
Exit	Heater	Stop heating command
Exit	Outlet valve	Open outlet valve command
Exit	Pump	Start pump command (to start emptying the tank)
Exit	Display	Display status command (‘Pumping’)

The size of this functional process is 5 CFP.

Functional process: Finish process

DM	Functional User / Object of interest	Data Group
Entry	Low level sensor	Low level reached message
Exit	Pump	Stop pump command
Exit	Outlet valve	Close outlet valve command
Exit	Display	Display status command (‘Finished’)
Exit	Audible alarm	Sound alarm command (to inform operator)
Exit	Clock	Stop clock command

The size of this functional process is 6 CFP.

(For polling the devices, ‘prompt message Entries’ are assumed (see Measurement Manual section 3.5.9., Rule a))

Functional process: Fault check

DM	Functional User / Object of interest	Data Group
Entry	Clock	Clock tick (to start fault check process)
Entry	Inlet valve	Inlet valve status (from polling)
Entry	Outlet valve	Outlet valve status (from polling)
Entry	Heater	Heater status (from polling)
Entry	Pump	Pump status (from polling)
Exit	Audible alarm	Start alarm command (if device fault(s) detected)
Exit	Display	Display faulty device(s) command (if there is a fault[2])

The size of this functional process is 7 CFP.

The total software functional size of the PLC software is 4 + 4 + 5 + 6 + 7 = 26 CFP.

4.1.3 Measurement of a change to the PLC software

Requirements

It has been decided to remove the audible alarm device and to adapt the software accordingly.

Analysis

The commands from the software to the alarm device can be removed, i.e. the Exit data movements of the audible alarm data group in the last two functional processes must be removed. The functional size of the change is 2 CFP. The resulting software functional size will be 24 CFP once the change is made.

4.2 Timing functionality

(See the Glossary for the definitions of ‘clock’ and ‘timer’ as used in this Guideline.)

Measuring timing functionality requires clear specifications on what functions are allocated to the hardware part of the functionality, and what are specifically allocated to the software part.

EXAMPLE 1. Timing functionality needed, for example, to control a pre-set time interval can be implemented in several ways, with different divisions between the hardware and software:

- *A hardware clock generates pulses (‘clock ticks’) at regular defined intervals each of which triggers a functional process of the software. The software keeps track of the pulses, may convert them to seconds or minutes if needed, and increments the elapsed time until the pre-set time is reached.*
- *A hardware timer both generates and keeps track of the pulses and transforms them into seconds, minutes etc., in an internal register if needed. The software may start the timer which informs the software when the desired time is reached. This mechanism is used in the next Example 2.*

EXAMPLE 2. A web-server must access a customer information system to retrieve some customer data. In addition to handling this request, the server starts a monitoring process to check that the request for customer information is handled within a set time. The aim is

[2] In this case, the display is shown on the context diagram as the functional user (not the human operator that reads the display). This Exit is therefore not an ‘error/confirmation message’, as defined in the Measurement Manual.

to ensure that the human user who seeks the customer information is not left hanging indefinitely if the customer information system fails to respond. Figure 4.2 shows a message sequence diagram for a simple example (no re-tries) of how this might be done via the interactions of the functional processes of the four participants, which are functional users of each other:

- *Web-server (functional user 1)*
- *Customer information system (functional user 2)*
- *Monitor (functional user 3)*
- *Real-time Timer (functional user 4)*

The web-server, after issuing the request to the customer information system, issues another message to the monitor, requesting it to respond if the given time-out period is exceeded. If the web-server receives the data from the customer information system within the time-out period, it tells the monitor to stop monitoring. Otherwise, if the web-server first receives a reply from the monitor that the time-out period has passed, the web-server issues a time-out message to the functional user that requested the customer data.

The monitor logs the request from the web-server and issues a request to the real-time timer, asking for a response within the given time-out period. (The timer may be implemented in hardware and/or software of the RTOS; it does not matter.) The monitor next receives either a message from the web-server to stop monitoring, or a message from the timer that the time-out period is complete. If the latter, the monitor sends a time-out message to the web-server. On completion, the monitor cancels the request from its log.

In Figure 4.2, the data movements of the timing functionality are shown as red dashed lines. The functionality requires 4 CFP for the web-server to request the monitoring function, in addition to the 2 CFP to obtain the customer data. The monitor requires 8 CFP to fulfill its requirement. (The 'delete request' Write is counted only once, although it may be issued at two alternative times, depending on whether the customer data are returned within the given time-out period or not.)

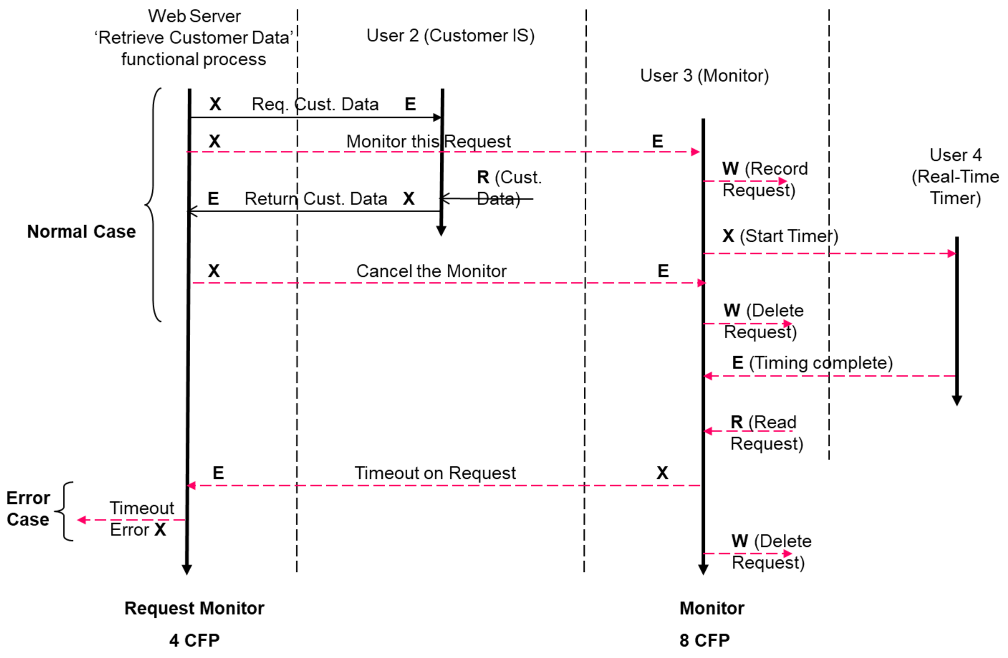

Figure - 4.2 The functionality for a web-server to monitor 'time-out'

4.3 Intruder alarm system

Outline statement of requirements

This case concerns a domestic intruder (or burglar) alarm system. Its main purpose is, when it is activated, to start one or two sirens (devices that make a loud noise) if a sensor detects a movement inside the house or if the front door is opened.

We do not have a statement of requirements, so we deduce the functionality available to normal house occupants and allocated to software from knowing how to use the system and by examining it physically. We are not interested in the functionality provided for the alarm maintenance engineer, nor in the functions to set-up the system when it is first installed.

The software supports the alarm system's human interface via a keypad and red/green LED's. The software also accepts data from a device that can sense whether the main front door of the house is open or not, and from several internal movement detectors. (The alarm system can handle any number up to 10 movement detectors. The number does not matter for this analysis as they are all identical and equivalent.) The alarm system also controls an internal and an external siren.

The alarm system is always powered 'on', but is not 'active', i.e. the movement detectors and the front door sensor are not working, unless the system is activated by the house occupant (the person normally resident in the house). When the system is activated, either the software waits in a state where it can receive signals from these sensors, or the software polls the sensors to obtain their state. We do not know which process is used and it does not matter for the functional size measurement.

To activate and de-activate the alarm system, the house occupant must enter the correct PIN (Personal Identification Number) within a pre-set time. The PIN is stored by the software and can be changed, so there must be some persistent storage. When the first digit of a PIN is entered, the internal siren is started; this siren is stopped on entry of all digits of the correct PIN. If the wrong PIN is entered three times or if the correct PIN is not entered within the pre-set time, the external siren is also started.

There is a battery to provide continuity if the mains electricity power supply fails, so there must be a power voltage detector.

The green LED is illuminated when power is switch on. If a siren is started or if the mains power fails, the green LED is switched off and the red LED is illuminated.

As certain functions must be completed within pre-set times, there must be a clock or timer mechanism. For example, if the alarm system is activated before leaving the house, the occupants must leave and close the front door within a pre-set number of seconds; if not, the sirens are started. The external siren must not continue for more than the legal limit of 20 minutes.

We do not know how the clock/timer is implemented but assume a software implementation for simplicity, which starts whenever needed. The functionality to keep track of elapsed times is then a form of data manipulation, which we can ignore.

Measurement strategy parameters

Purpose of the measurement: To measure the functional processes of the embedded application software available to the house occupant for normal operation.

Measurement scope: The alarm system embedded application software functions available to the house occupant for normal operation. (We are not interested if there is an operating system)

Functional users: A context diagram shows the hardware functional users and how they interact with the software. Note that the movement detectors are all functionally identical, so do not need to be distinguished. The human user of the alarm system, referred to as 'the occupant' is not a functional user; he/she interacts with the application only via the keypad and the audible and visual signals.

Layer: Application.

Level of decomposition: 'level 0', i.e. no decomposition.

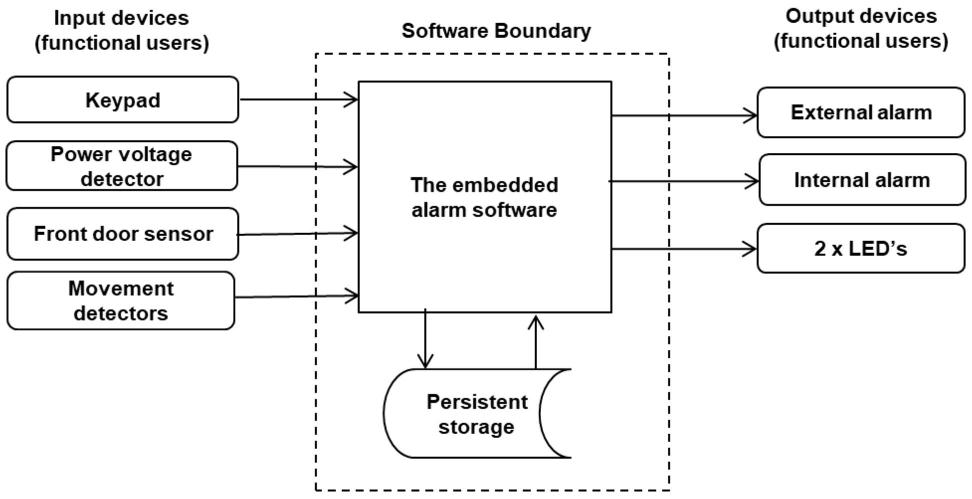

Figure 4.3 – The Intruder Alarm System Context Diagram

The functional processes: After initial set-up, the alarm system application provides the occupant with nine functional processes. These can be identified by considering the events that the software must respond to.

1) *The occupant wishes to change the existing PIN.*
2) *The occupant wishes to leave the house and activate the alarm system.*
3) *The front door sensor detects that the door has been opened whilst the alarm system is activated.*
4) *The occupant wishes to activate the alarm system whilst he/she is in the house, e.g. when retiring at night, out of range of the movement detectors.*
5) *The occupant wishes to deactivate the alarm system when inside the house, e.g. when getting up in the morning before moving within range of the movement detectors.*
6) *A movement detector signals a movement whilst the alarm system is activated (which starts the internal siren).*
7) *The occupant wishes to cancel the siren(s) and to deactivate the alarm system by entering the correct PIN following events 3) or 6).*
8) *The power voltage detector signals failure of the mains electrical supply*
9) *The power voltage detector signals restoration of the mains electrical power supply.*

Analysis of an example functional process:

We analyze the event 3) on the list above (the front door is opened whilst the alarm system is activated). When the front door sensor detects this event, the internal siren starts; the correct PIN code must then be entered within a pre-set time to de-activate the system and to stop the internal siren. If the PIN code isn't entered before the pre-set time, or the wrong code is entered more than three times, the external siren also starts. The functional process has the following data movements.

Functional process: Possible intruder detected. Triggering event: Door opens whilst alarm system is activated.

DM	Functional User / Object of interest	Data Group
Entry	Front-door sensor	‘Door open’ message (triggering Entry)
Read	- / Occupant	PIN (from persistent storage)
Exit*	Green LED	Switch ‘off’ command
Exit*	Red LED	Switch ‘on’ command
Exit	Internal siren	Start noise command
Entry	Keypad	PIN (If the wrong code is entered, the user may enter the PIN two more times but the process is always the same so it is only measured once.)
**	Green LED	Switch ‘on’ command (after successful entry of PIN)
**	Red LED	Switch ‘off’ command
Exit	Internal siren	Stop noise command (after successful entry of PIN)
Exit	External siren	Start noise command (after three unsuccessful PIN entries, or if the PIN is not entered in time)
Exit	External siren	Stop noise command (after 20 minutes, a legal requirement)

NOTE: : (*) The green and red LEDs are different types as they are subject to different functional user requirements, therefore identify two functional user types. (**) These are repeat occurrences of the Exits to the LED’s earlier in the process, but with different data values (‘on’ instead of ‘off’, and vice versa).

The total size of this functional process is 9 CFP

4.4 Cooker software defined as a finite state machine

An extremely simple cooker can be set to cook for multiples of one minute, provided its door is closed.

Requirements

- When the power is switched on, the cooker software can receive input from the door and from a start button, and can send signals to switch an internal light, and the heater, on or off. The software can also send signals to a timer to set the cooking time and can receive a signal from the timer when cooking is complete.
- Cooking starts with pressing the start button provided the door is closed. If the door is open pressing the start button has no effect.
- Opening the door during cooking turns the heater off.
- Whilst cooking or whilst the door is open, the cooker light is on.
- The cooking time is set in multiples of a minute.
- Each time the start button is pushed adds one minute to the cooking time.
- When the timer stops, either because the door is opened whilst cooking is in progress, or because the timer signals that cooking is completed, the timer resets itself to zero.
- The initialization of the cooker software is out of the scope of this case. Assume the power is on and the cooker is in a ‘standby’ state.

Context diagram

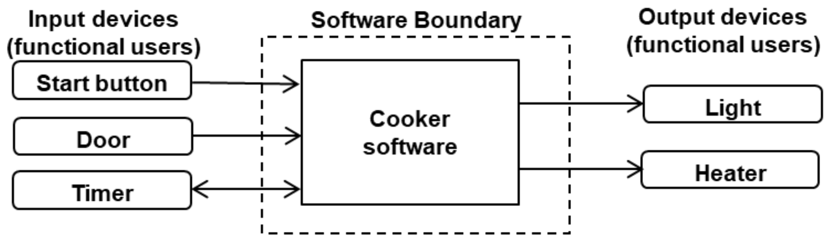
Input devices (functional users)
Start button
Door
Timer
Software Boundary
Cooker software
Output devices (functional users)
Light
Heater

Figure 4.4 – Cooker Context Diagram

The state transition diagram of the cooker is shown in Figure 4.5. Boxes represent states and arrows represent the transitions from one state to another (possibly the same state). The events which cause the cooker to move between its states are triggering events. These are prefixed by 'TE' and the functional users that sense the events by 'FU'.

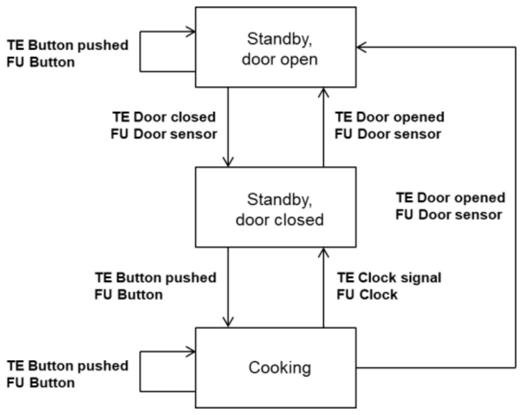
TE Button pushed FU Button
Standby, door open
TE Door closed FU Door sensor
TE Door opened FU Door sensor
Standby, door closed
TE Door opened FU Door sensor
TE Button pushed FU Button
TE Clock signal FU Clock
TE Button pushed FU Button
Cooking

Figure 4.5 – Cooker State Transition Diagram

Analysis

The functional users of the cooker software on the input side are the door sensor and the push button. On the output side the functional users are the cooker light and the heater. The functional user that is on both the input and the output side is the timer. As noted in section 3.2.1, in this case the object of interest of each data group entering the software is also the functional user that sent the group (i.e. the functional user is sending data about itself) and similarly the object of interest of each data group that leaves the software is also the functional user that receives the group (i.e. the functional user is being sent data about itself).

The events that actually trigger the software to start a functional process are as follows. As there is here a one-one correspondence between triggering events and functional processes, the same name is used for both.

Table 4.2 - Cooker, triggering events and functional processes

Triggering event	Functional user that initiates the functional process	Functional process
Door closed	Door sensor	Door closed
Button pushed	Push button	Button pushed
Timer signal (cooking ended)	Timer	Timer signal (cooking ended)
Door opened	Door sensor	Door opened

The functional processes of the cooker are as follows:

Functional process: Door closed

DM	Functional User / Object of interest	Data Group
Entry	Door sensor	Door closed signal (triggering Entry)
Exit	Cooker light	Switch ‘off’ command

The size of this functional process is 2 CFP.

Functional process: Button pushed

DM	Functional User / Object of interest	Data Group
Entry	Start button	Button pushed signal (triggering Entry)
Entry	Door sensor	Get door status
Exit	Heater	Heater ‘on’ command (if door closed)
Exit	Cooker light	Light ‘on’ command (if door closed)
Exit	Timer	Start or increment cooking time command (each push adds one minute to the cooking time if door is closed)

The size of this functional process is 5 CFP.

Functional process: Timer signal (cooking ended)

DM	Functional User / Object of interest	Data Group
Entry	Timer	Timing stopped signal (triggering Entry)
Exit	Heater	Switch ‘off’ heater command
Exit	Cooker light	Switch ‘off’ cooker light command

The size of this functional process is 3 CFP.

Functional process: Door opened

DM	Functional User / Object of interest	Data Group
Entry	Door sensor	Door open signal (triggering Entry)
Exit	Cooker light	Switch ‘on’ cooker light command
Exit	Heater	Switch ‘off’ heater command
Exit	Timer	Stop timer command

The size of this functional process is 4 CFP.

The total functional size of the cooker software in the scope is 2 + 5 + 3 + 4 = 14 CFP.

Discussion

Note an important point about interpreting state transition diagrams. Not all state transitions correspond to separate functional processes. In this example there are seven state transitions but only four functional processes. Only events detected by or generated by a functional user external to the software can trigger a functional process. Each functional process must deal with all states and state combinations that it can encounter when responding to a given triggering event.

As an example, the triggering event ‘button pushed’ can occur when the cooker is in each of the three states. The event of the button being pushed takes place in the external world of the hardware and is entirely independent of the state of the machine. The one functional process that must handle the ‘button pushed’ event responds in three ways dependent on the state of the machine at the time the button is pushed namely:

- In the ’standby, door open’ state, it does nothing, i.e. it stops after having found that the door is open;
- In the ‘standby, door closed’ state, it sends signals to start the heater and switch on the light, and to start the timer for one minute of cooking;
- In the ‘cooking state’, it executes the same data movements as in the previous state but since the heater has already started and the light is already on, the effect is only to add one minute to the total cooking time.

In this example, we have assumed that the cooker can perform its functions by simply checking if the door is open or closed. In a more complex case, software may need to record the state of the machine and to update it in persistent storage every time the state changes. This would avoid the need for the software to determine the state of the machine each time a new event is signaled.

Similarly, the ‘door opened’ event can occur when the machine is in two states. The one corresponding functional process must deal with the two states.

4.5 Tire-pressure monitoring system

Requirements

- A tire-pressure monitoring system (TPMS) monitors the pressure of each of the four tires of a car.
- Each wheel has a sensor which obtains the pressure of its tire.
- As soon as the car’s electrical power supply is turned on, a clock activates the TPMS software once per second to retrieve the status of the four sensors, whether the car is moving or not. The sensors return their status, consisting of the sensor id (which identifies the particular wheel) and tire pressure.
- If the pressure is too low or too high - the values are in the software - the TPMS turns on the relevant red warning LED(s) at the dashboard (the sensor location is therefore relevant).
- If the pressure becomes normal again, the TPMS switches off the relevant red warning LED(s) at the dashboard.
- The TPMS electronic control unit (ECU), the clock, the tire pressure sensors and the dashboard LED’s are coupled by a CAN-bus (CAN = controller–area network).

The purpose of the measurement is to size the functionality of the TPMS.

Context diagram

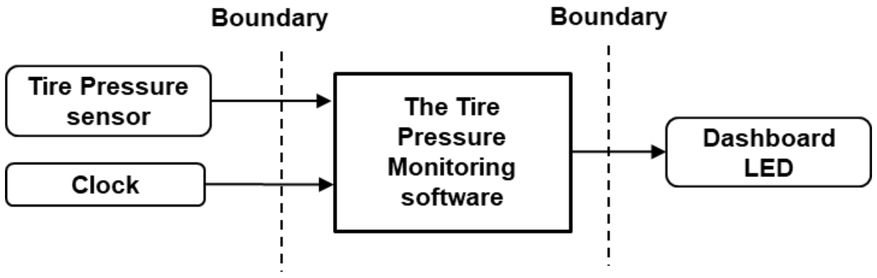

Figure 4.6 – TPMS, context diagram

Analysis

The four sensors are subject to the same FUR (‘obtain tire pressure’), therefore one functional user must be identified that represent these four occurrences: ‘Tire pressure sensor’ (short: Sensor). The same applies to the four LEDs (FUR: ‘turn on/off’), i.e. one functional user ‘Dashboard LED’ (short: LED). The third functional user is the clock.

The Sensor functional user sends the sensor ID and its value, the clock sends the clock signal, the LED functional user receives the LED ID and ‘on’ or ‘off’, all data that describe the functional user concerned. The three functional users are therefore objects of interest.

The CAN-bus controllers form a collection of software that together provides a cohesive set of services that the TPMS software can use and are therefore in a software layer that is separate from the layer in which the TPMS software resides. The network controllers are therefore not in the scope of this measurement. Note that if they were within the scope of the measurement, they must be measured separately as the controllers are software in another layer.

The software must respond to one triggering event, the clock signal that is sent every second, so consists of one functional process:

Table 4.3 - TPMS, triggering events and functional processes

Triggering event	Functional user that initiates the functional process	Functional process
Clock signal	Clock	Start TPMS software

There is no requirement to store or retrieve any persistent data. The table shows the data movements, the data groups moved for the functional process. Further explanation follows.

Functional process: Start TPMS software

DM	Functional User / Object of interest	Data Group
Entry	Clock	Start monitoring signal (triggering Entry)
Entry	Tire pressure sensor	Obtain tire pressure
Exit	Warning LED	Switch ‘on/off’ LED (if needed)

The size of this functional process is 3 CFP.

This simple case illustrates an important aspect of many real-time systems that have multiple occurrences of sensors or output devices. Examples would be multiple sensors (all with or without IDs) distributed across a sheet of material passing through a set of rollers controlled by a process control system. Where such sensors (or output display devices) are subject to the same FUR, identify one functional user for the occurrences of the sensors or devices.

4.6 Automation of sizing real-time requirements

Automation of the measurement of requirements for real-time embedded software of vehicle Electronic Control Units modeled with the Matlab Simulink tool is described in [7][7] and in the PhD thesis of Hassan Soubra [8]. A concise English description of the method, copyright Renault, is available from the download section of www.cosmic-sizing.org [9].

Automation of the measurement of requirements expressed in UML (not specifically of real-time software) is described in [10].

4.7 Measurement of data manipulation-rich real-time software

An example in this section illustrates that the assumption is reasonable that data manipulation functionality (or ‘algorithms’) of real-time software can be accounted for by the COSMIC method. The example does not, of course, prove that the assumption is always reasonable.

For ways in which to deal with software for which it is known that certain areas of the functionality have a high concentration of data manipulation, see section 3.2.3 of this Guideline.

The distribution of algorithms in some avionics software

A large component of the software of a very complex real-time avionics system was measured using the COSMIC method [11][11]. The total size of the requirements (held in a modelling tool) for the component was over 8000 CFP. Implementation required over 80,000 lines of source code in the Ada language.

This one system component consisted of 33 sub-components. Within each sub-component, the number of lines of Ada code associated with each data movement was also counted. This is known as the 'NOLA', for 'number of lines of algorithm'. Hence the 'NOLA per Data Movement' could be calculated for each of the 8000+ data movements.

Figure 4.8 shows a histogram of the frequency of the 'NOLA per Data Movement', for all except five of the data movements. The five data movements with exceptionally high NOLA had 28 (x2), 36, 40 and 138 NOLA. (Example: the histogram shows that 20 of the 8000+ data movements had seven NOLA.)

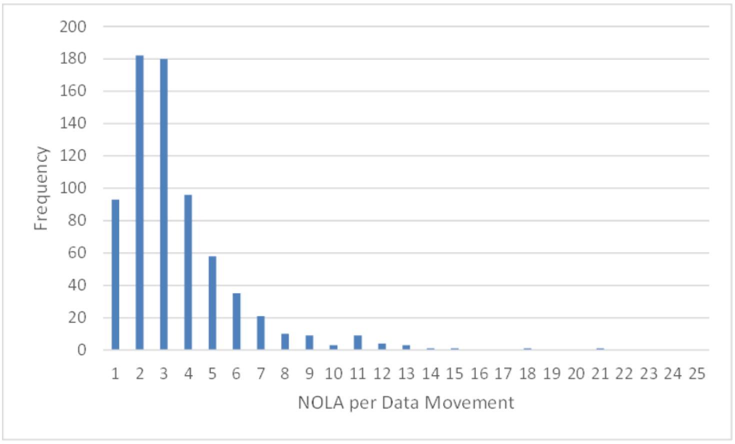

Figure 4.8 - Frequency of NOLA per Data Movement' [11]

The following parameters were derived from these data:

Parameter	Value
Median NOLA per Data Movement	2.4
Mean average NOLA per Data Movement	3.5
Data Movement upper size limit which accounts for 95% of the total NOLA	8 CFP
Data Movement upper size limit which accounts for 99% of the total NOLA	14 CFP

The data and the analysis indicate that the NOLA per Data Movement values have a limited range, apart from a very few exceptions. This finding supports the COSMIC method assumption that a count of data movements reflects the amount of data manipulation and thus is a good reflection of the functional size, at least for this particular piece of real-time software.

4.8 Sizing the memory requirements of vehicle Electronic Control Units

A paper entitled ' On the Conversion between the Sizes of Software Products in the Life Cycle' by C. Gencel, R. Heldal and K. Lind, presented at the International Workshop on Software Measurement in Stuttgart, November 2010, describes the application of the COSMIC method to size the software embedded in Electronic Control Units of Saab cars, manufactured in Sweden. The purpose of the study was to examine the relationship between the COSMIC-measured functional size and the resulting memory space needed by the object code, measured in bytes. An extremely good linear correlation was found.

In the paper, the authors state: ‘This paper shows that it is possible to obtain accurate code size estimates even for software components containing complex calculations, as long as the components contain similar complexity proportional to the number of component interfaces.’

Renault [12] also reported a good correlation of code size in bytes versus COSMIC-measured functional size in units of CFP, as in the graph below.

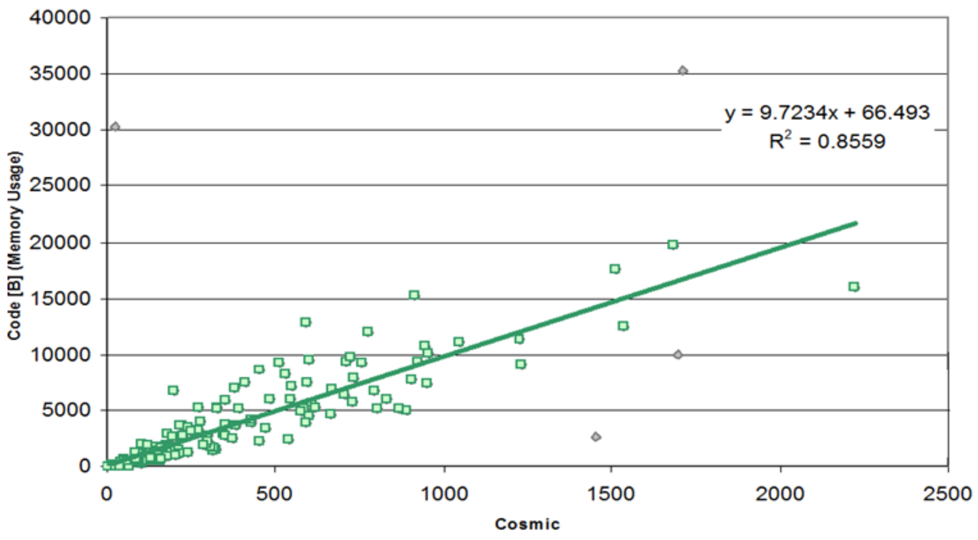

Figure 4.9. Code size (bytes) versus COSMIC functional size (CFP) [12]

REFERENCES

All the COSMIC documents and the documents indicated by (*) listed below, including translations into other languages, can be obtained from the download section of www.cosmic-sizing.org.

[1] COSMIC, Measurement Manual, v4.0.2, December 2018 (*)

[2] ISO 14143:2007 Software Engineering – Software Measurement – Functional Size Measurement, Part 5, Determination of Functional Domains for Use with Functional Size Measurement.

[3] Toivonen, H., Defining measures for memory efficiency of the software in mobile terminals International Workshop on Software Metrics, 2002.

[4] COSMIC, Guideline for 'Measurement Strategy Patterns (*)

[5] COSMIC, 'Guideline on Non-Functional & Project Requirements and Constraints' (*)

[6] COSMIC, 'Guideline for Early or Rapid COSMIC Functional Size Measurement' (*)

[7] Soubra, H., Abran, A., Stern, S., Ramdan-Cherif, A., Design of a Functional Size Measurement Procedure for Real-Time Embedded Software Requirements Expressed using the Simulink Model. IWSM-MENSURA, Nara, Japan, 2011.

[8] Soubra, H., PhD thesis Automation de la mesure fonctionnelle COSMIC-ISO 19761 des logiciels temps-réel embarqué, en se basant sur leurs specifications fonctionnelles. Ecole de technologie supérieure (ETS), Université du Québec and Université de Versailles at St-Quentin, in collaboration with Renaults SAS.

[9] Renault, COSMIC Rules for Embedded Software Requirements Expressed using Simulink, 2012 (*).

[10] Swierczek, ,J,, Automatic COSMIC sizing of requirements held in UML, COSMIC Masterclass, IWSM 2014, Rotterdam (*)

[11] Private client data.

[12] Stern, S., Gencel, C., Embedded software memory size estimation using COSMIC: a case study, IWSM 2010 (*)

REAL-TIME DOMAIN GLOSSARY

Actuator. A device that converts an electrical signal to a mechanical movement.

Clock: (as used in this Guideline) A hardware device that generates a stream of pulses at constant frequency.

Control. Directing or guiding

Dumb device. Any type of device that sends its data automatically on receipt of a prompt message.

NOTE: For measurement purposes, an Entry data movement to the software needing the data suffices to obtain the data from a dumb device.

Intelligent device. Any type of device that must be told explicitly what data is required in order for it to produce the required output data.

NOTE: For measurement purposes, an Exit data movement from the software specifying the required data is received by an intelligent device as an Entry. The device issues an Exit conveying the requested data which is received by the requesting software as an Entry.

Monitor. Keeping a check on something

Sensor. A device that converts a physical signal to an electrical signal that is made available to the software in the form of a data group.

Timer: (as used in this Guideline) A hardware device or software process that can measure a time duration.

APPENDIX

1 Acknowledgements

Reviewers and Editors (*) of version 2.0 of this Guideline			
Alain Abran* École de Technologie Supérieure, Université du Québec, Canada	Peter Fagg Pentad United Kingdom	Cigdem Gencel, Free University of Bozen/Bolzano, Italy	Arlan Lesterhuis* MPC Netherlands
Bruce Reynolds Tecolote Research USA			

2 Version Control

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

DATE	REVIEWER(S)	Modifications / Additions
June 2012	COSMIC Measurement Practices Committee	First version 1.0 issued
April 2015	COSMIC Measurement Practices Committee	Version 1.1 brought in line with the Measurement Manual v4.0.1. See Appendix A for changes made
November 2016	COSMIC Measurement Practices Committee	Version 1.1.1 has various improvements for ease of understanding. See Appendix A for changes made.
November 2018	COSMIC Measurement Practices Committee	Version 1.1.2 has a few changes to comply with the Measurement Manual v4.0.2, further only minor changes were needed. See Appendix 3 for changes made.
November 2019	COSMIC Measurement Practices Committee	Non-essential sections removed. Change History of previous versions moved to a separate document, to be stored in cosmic-sizing.

3 Guideline Change History

This section contains a summary of the principal changes made in the evolution of this 'Guideline for sizing real-time software' from version 1.1.2 to the version 2.0.

V2.0 Ref	Change from version 1.1.2 to version 2.0
General	The Guideline Change History has been limited to the changes that have been made from the previous version 1.1.2. Changes to older versions have been moved to a separate document that will be added shortly to the Knowledge base of cosmic-sizing. Some minor editorial changes have been made to increase readability.
1.2.2, 1.2.3, 1.2.4, 1.2.5, 1.2.6	These sections on the requirements in the EARS syntax, requirements for a finite state machine, requirements for a programmable logic controller, requirements in specialized tools, requirements in UML and on non-functional requirements have been removed as not being essential.

1.2.7	A large part of this section on non-functional requirements repeated text of the Guideline on Non-Functional & Project Requirements and is removed, the remaining text becoming section 1.2.2.
4.3	The red and green LEDs are different functional user types, explanation added to the Intruder alarm system example.
Reference section	Updated to remove documents referred to in the deleted sections above

4 Change requests, comments, questions

Where the reader believes there is a defect in the text, a need for clarification, or that some text needs enhancing, please send an email to: mpc-chair@cosmic-sizing.org
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.