COSMIC Measurement Manual for ISO 19761

Part 3b: Real-time Examples

August 2021

Foreword

This document consolidates the examples from the previous Real-time Guideline and from Part 3 of the Measurement Manual. The examples are ordered per COSMIC phase and should speak for themselves, therefore this document does not contain any explanation of the COSMIC method, if needed consult the COSMIC Measurement Manual. Also, the real-time related text has been omitted as it seems superfluous for the intended audience.

The COSMIC Measurement Manual describes the core measurement method. This version 5.0 consists of parts:

Part 1: Principles, definitions & rules* (17 pages)

Part 2: Guidelines* (18 pages)

Part 3: Examples of COSMIC concepts and measurements, consisting of:

Part 3a	Standard Measurement Strategy Examples (13 pages)
Part 3b	Real-time Examples (32 pages)
Part 3c	MIS Examples. (58 pages)

* Parts 1 and 2 describe the entire material necessary for certification.

Further explanations, guidelines, translations, practical examples and other publications are available from www.cosmic-sizing.org.

August 2021 minor editing:

In the Foreword the names of the Parts have been updated. No other changes.

Editors:

Alain Abran, Ecole de technologie supérieure – University of Quebec (Canada),
Peter Fagg, Pentad (UK),
Arlan Lestherhuis (The Netherlands).

Other members of COSMIC Measurement Practices Committee:

Table of Contents

1. THE MEASUREMENT STRATEGY PHASE.

1.1 Purpose and scope.

For examples of purposes and scopes see chapter 3 Comprehensive Examples and the COSMIC Case studies (freely available at cosmic-sizing.org).

1.2 Functional User Requirements (FUR)

EXAMPLE: 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.

1.3 Non-functional requirements

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 an 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 also [5]).

1.4 Types versus occurrences.

EXAMPLE 1: 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.

EXAMPLE 2: Suppose a functional process that must control the temperature of an oven once every ten seconds. The functional process will be executed, i.e. it will occur once every 10 seconds. During its execution, the Exit data movement of the process to switch the heater on or off may or may not be executed i.e. it may or may not occur at all in any cycle, depending on whether the heater must be switched on or off, or left in its current state. The Exit data movement is counted once in the functional process, regardless of whether or not it occurs in a particular execution.

1.5 Layers.

1.5.1 A layered architecture of embedded real-time software.

EXAMPLE 1: The physical structure of a typical layered software architecture supporting a piece of embedded real-time software is given in Figure 1.1. (Note: simple uni-tasked real-time embedded software may not need a real-time operating system.)

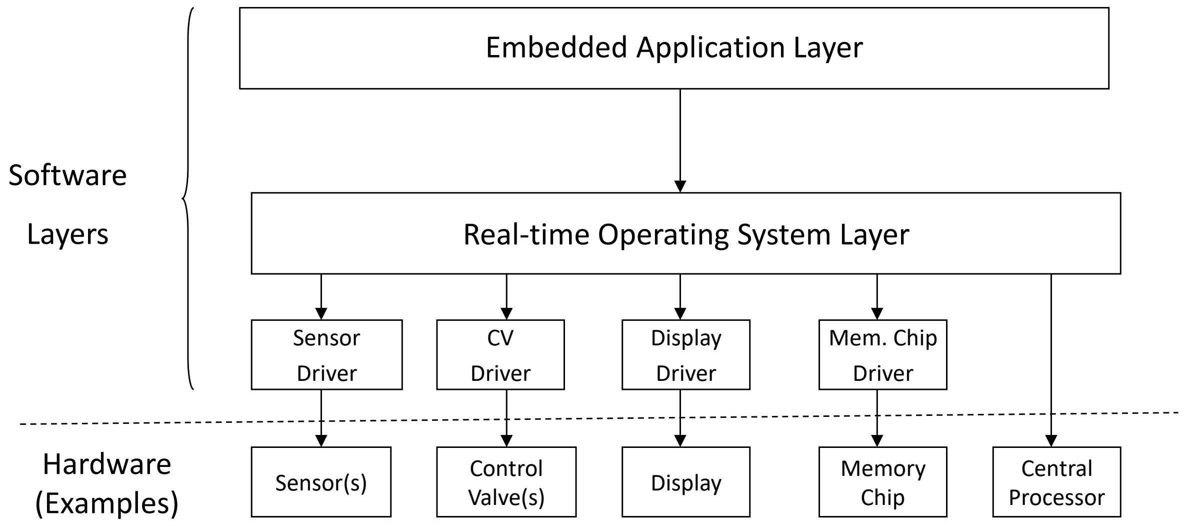


Figure 1.1 - Typical layered architecture for a real-time embedded-software system.

1.5.2 Layered architectures in industry

EXAMPLE 2: 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.

EXAMPLE 3: The 'AUTOSAR' architecture of the automotive industry that exhibits all the different types of correspondence rules between layers now described in the principles for a layer.

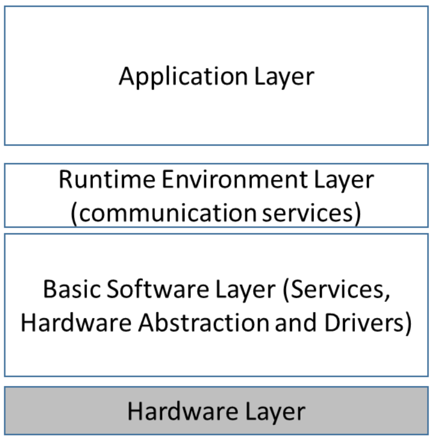

Application Layer
Runtime Environment Layer (communication services)
Basic Software Layer (Services, Hardware Abstraction and Drivers)
Hardware Layer

Figure 1.2 – The structure of the AUTOSAR architecture

1.6 Functional users.

1.6.1 Typical functional users.

EXAMPLE 1. When measuring real-time software, the functional users that interact with the software being measured will be typically any of the following:

- a clock or timer,
- 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.

EXAMPLE 2. Section 3.1 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 3. The embedded software of a mobile device (tablet, smart 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 device 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.

1.6.2 Types of functional users.

EXAMPLE 1. 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).

EXAMPLE 2: Each wheel of a car has a sensor that 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 indicated on a diagram of the four wheels on a display screen at the dashboard. The functional users are the four sensors and the four indications on the display screen. However, the four sensors are subject to the same requirement (and idem for the indications on the screen), so identify one functional user type 'sensor' and one functional user type for the indications on the display screen.

1.6.3 Incompatibility of functional users

EXAMPLE. Consider the embedded software of a copier. 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[1]. Do NOT try to mix the two views; a size measurement from a 'mixed' human/hardware view would be very difficult to interpret.

1.7 Levels of decomposition and granularity.

EXAMPLE 1: A group of functional users might be a 'control panel' that has many types of instruments, or 'central systems'. A group of events might be indicated in a statement of functional requirements at a high level of granularity by an input stream to an avionics software system labelled 'pilot commands'.

EXAMPLE: For an example of sizing at varying levels of decomposition and granularity, see the telecoms system example in the Early Software Sizing with COSMIC: Experts Guidelines [6].

[1] 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.

1.8 Context diagrams.

EXAMPLE: Figure 2.3 shows the context diagram for a simple intruder alarm embedded software system.

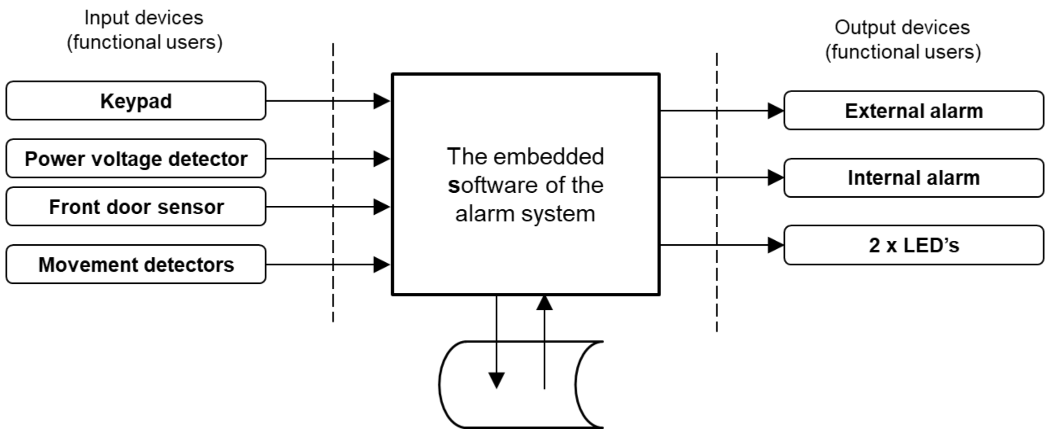

Input devices (functional users)
Keypad
Power voltage detector
Front door sensor
Movement detectors
The embedded software of the alarm system
Output devices (functional users)
External alarm
Internal alarm
2 x LED's

Figure 1.3 - Context diagram for the embedded software of an intruder alarm system.

2. THE MAPPING PHASE.

2.1 Triggering events and functional processes

2.1.1 Identifying a functional process.

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 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.

EXAMPLE 4: 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).

EXAMPLE 5: 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.

EXAMPLE 6: 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).

EXAMPLE 7: 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.

2.1.2 A clock triggering a functional process

EXAMPLE 1: 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.

EXAMPLE 2: 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.

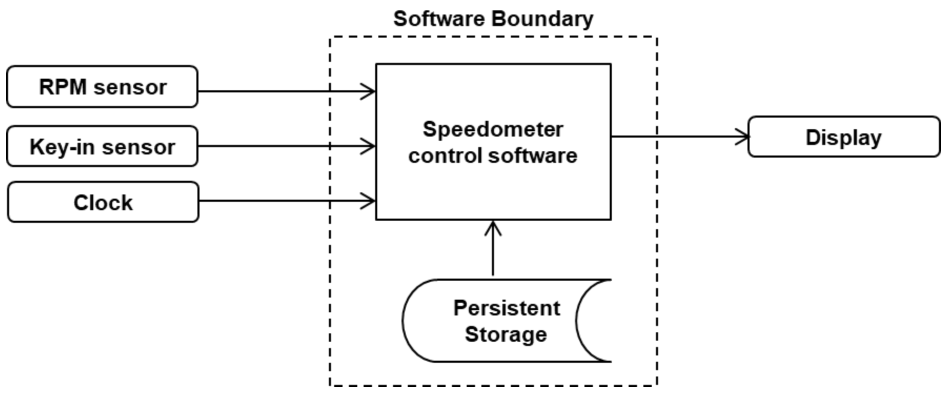


Figure 2.1 – Context diagram for the speedometer software

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.

2.1.3 Different processing paths, one functional process.

EXAMPLE: One triggering Entry (aircraft altitude information sent by the Geographical Positioning System) to a functional process of an avionics system will lead to 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.

2.2 Objects of interest and data groups

2.2.1 The functional user as object of interest

EXAMPLE 1. The speedometer control software in Example 2 of section 2.1.2 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) is also the object of interest of a data group that it sends (i.e. it is sending data about itself).

EXAMPLE 2: A data group entering software from a physical device informs about the current state of the device. In this case the device is the object of interest (and functional user) and the data group conveys its state, such as that a valve is open or closed, leading to the start of a functional process. The physical device is the object of interest. Similarly, a data group output to a device, such as to switch a warning lamp on or off conveys state data about the lamp object of interest.

EXAMPLE 3: Suppose a temperature sensor A sends a measure of the current temperature of a material for processing by a functional process. The sensor provides information about its own state and is thus object of interest of the information data group.

2.1.2 Other examples of objects of interest

EXAMPLE 1: 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’.

EXAMPLE 2: A reference data structure, represents objects of interest whose attribute-values are given in 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.

EXAMPLE: 3: Files, commonly designated as ‘flat files’, represent objects of interest mentioned in the FUR, which are held on a storage device.

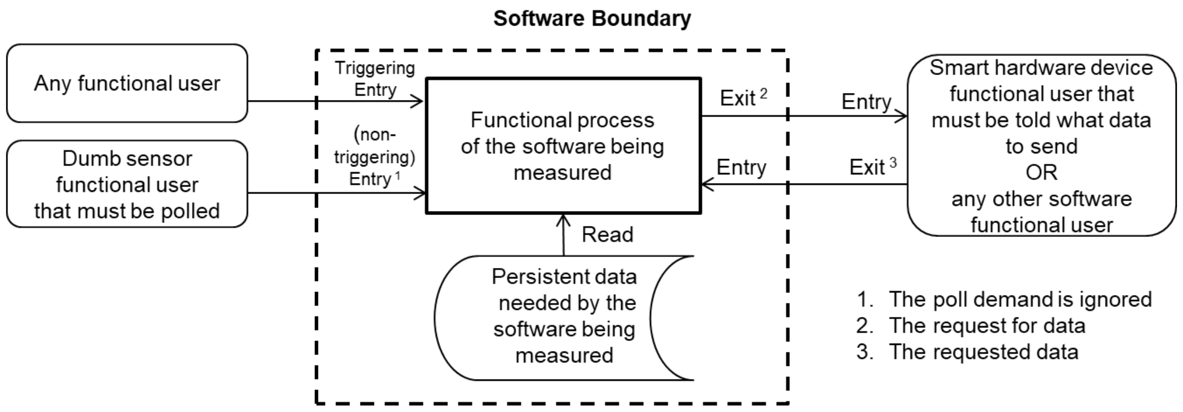
2.3 Data attributes.

EXAMPLE: A temperature sensor 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’.

2.4 Data movements.

2.4.1 The various ways of receiving or getting data

EXAMPLE 1. Figure 2.2 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
Dumb sensor functional user that must be polled
Triggering Entry
(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 2.2 - The various ways in which a functional process can receive or get data

2.4.2 Identifying data groups and data movements

EXAMPLE 1: 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.

EXAMPLE 2: For a clock-tick event occurring every 3 seconds, identify an Entry moving a data group of one data attribute. The object of interest (and functional user) is the clock, the data group conveys the state of the clock.

2.4.3 Types and occurrences of functional users and data groups.

EXAMPLE 1: A functional process is required to accept different data groups from two different seismometers (functional users) each responding to the same event e.g. a test explosion. Identify two Entries.

EXAMPLE 2: 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 from the position in the string of data values sent from the array of sensors, The processing of data from all the sensors in the array is identical. Identify one functional user for all the sensors and one Entry for the data obtained from all the sensors by the functional process.

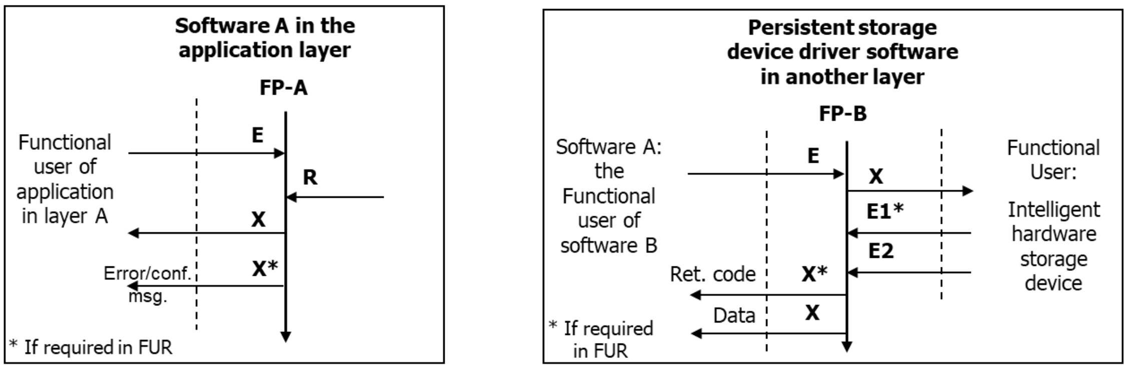
2.4.4 Data groups in a layered software architecture.

EXAMPLE 1: This example concerns the piece of software A in Figure 2.3 that is required to retrieve a stored data group. Consider a separate piece of software B that is the device driver for the intelligent hardware storage which holds the data group that the software A is required to access. (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 1.1. 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 1.1. 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 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 2.3 (a) shows the COSMIC model of this enquiry. Figure 2.3 (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).


Software A in the application layer
FP-A
Functional user of application in layer A
E
R
X
Error/conf. msg.
X*
* If required in FUR
Persistent storage device driver software in another layer
FP-B
Software A: the Functional user of software B
E
X
E1*
E2
Functional User:
Intelligent hardware storage device
Ret. code
X*
Data
X
* If required in FUR

Figures 2.3 (a) and (b) – Solution for a Read issued by software A in the application layer to software B in the device driver layer.

Figure 2.3 (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 E2 in Figure 2.3 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 E1* in Figure 2.3 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, as Reads and Writes account for any associated reporting of error conditions. 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 Figure 2.3 b). For example, this Figure does not show the effect of the device driver measuring a timeout for non-response from the hardware.

EXAMPLE 2: Suppose a functional process of a real-time process control 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. (Since the sensors are identical one Entry is identified and counted.)

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 1.1. The functional processes of the process control software and of the device driver software for the dumb sensors would be as shown in Figures 2.4 (a) and (b) below.

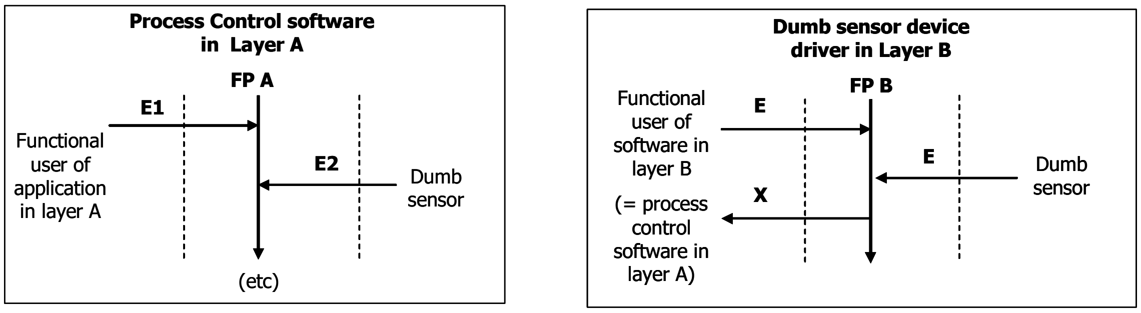

Process Control software in Layer A
FP A
E1
Functional user of application in layer A
E2
Dumb sensor
(etc)
Dumb sensor device driver in Layer B
FP B
Functional user of software in layer B
(= process control software in layer A)
E
X
E
Dumb sensor

Figure 2.4 (a) and (b) – Solution for a poll of dumb sensors.

Figure 2.4 (a) shows that the 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. (The device driver software is hidden at this level.)

Figure 2.4 (b) shows the model for the software that drives the dumb sensor devices. It receives data via an Entry from the process control software (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 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.

The apparent mis-match between the one Entry E2 from a dumb sensor to the process control 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.

2.5 Measuring the components of a distributed software system.

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 2.5 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.

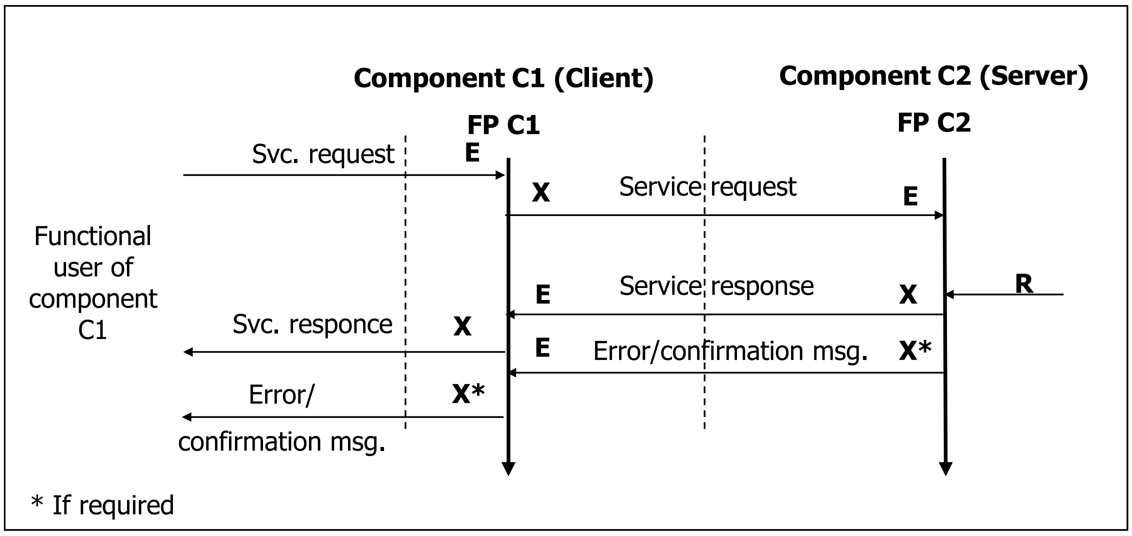


Figure 2.5 – Data exchanges between client and server components.

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 intermediate layers of their processors in a software architecture such as shown in Figure 1.1. 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.

Figure 2.5 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 enquiry therefore requires 6 data movements (i.e. 6 CFP) to satisfy the enquiry request for component C1 and 4 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 via a Read as shown in Figure 2.3 (a).

Component C2 will probably 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 Figure 2.3 (b).

2.6 Re-use of software.

EXAMPLE 1: 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). Measure this common functionality.as part of each functional process that requires it.

EXAMPLE 2: For a real-time application, the user that starts the application must be ignored. This user 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).

EXAMPLE 3: For a computer operating system, the user that starts the operating system is a bootstrap program that is started when the computer power is switched on. This user must be ignored.

EXAMPLE 4: 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 is being reused and should be ignored.

2.7 Error or fault messages

2.7.1 General examples

EXAMPLE 1: 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 S has failed’, where ‘S’ is a variable. This message should be identified as one Exit in that functional process to account for moving data about the functional user (and object of interest) sensor S.

EXAMPLE 2. The FUR of the software of Figure 2.4 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.

EXAMPLE 3: 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-to-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'

2.7.2 Error messages intended for hardware or software functional users.

EXAMPLE 1. 'In-line' error messages that are intended for hardware or software functional users. 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.

> Output= {VAL1, VAL2, VAL3, …, VALn, ERROR} where 'VAL' indicates a valid value.

EXAMPLE 2. Separate error messages that are intended for hardware or software functional users. 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. The general case is:

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

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

2.8 Changes to software.

See section 3.2 for an example of changes to software.

2.9 Data manipulation

EXAMPLE: 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. In COSMIC the two movements of data are considered to be multiple occurrences of the same Entry. Only one Entry may be identified for this data group for this functional process. Note that the calculation of the rate of change is associated with the Exit that reports the rate, there is no data manipulation associated with the two occurrences of the Entry..

3. COMPREHENSIVE EXAMPLES

3.1 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

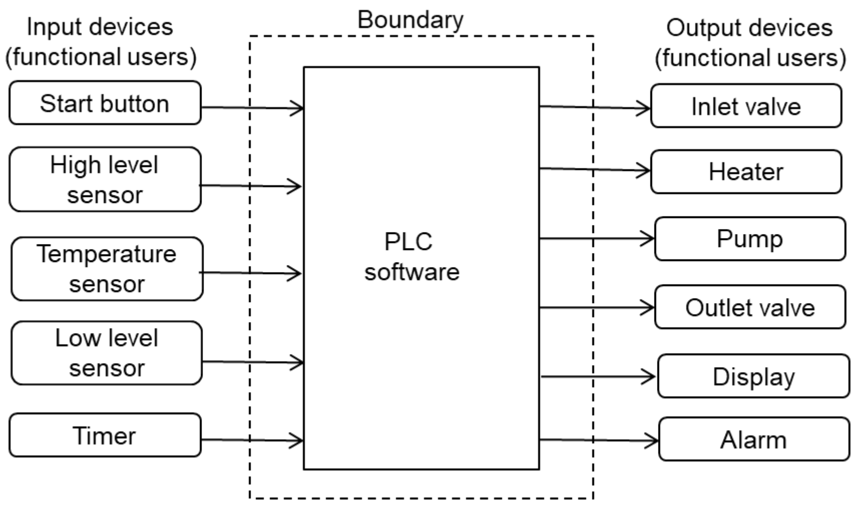


Figure 3.1 - Chemical Factory Process: PLC software Context Diagram

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). The triggering events and corresponding functional processes are:

Table 3.1 - 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

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.

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.

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.

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.

3.2 Measurement of a change to the PLC software

Requirements

In the Example of section 3.1 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.

3.3 Timing functionality

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.

[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.

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 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.)

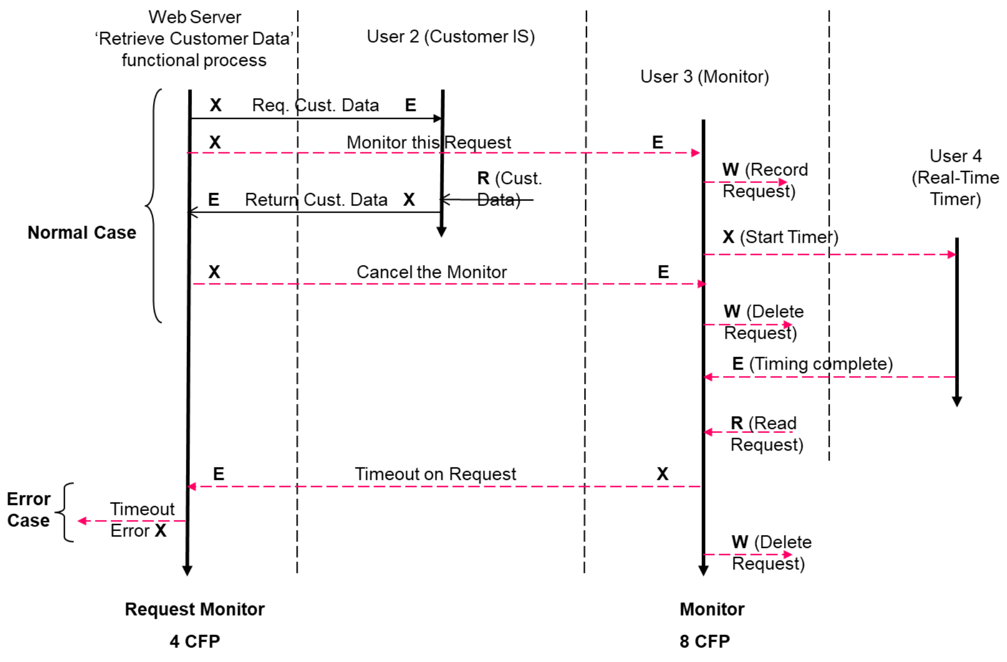


Figure - 3.2 The functionality for a web-server to monitor 'time-out'

3.4 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 layer.

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

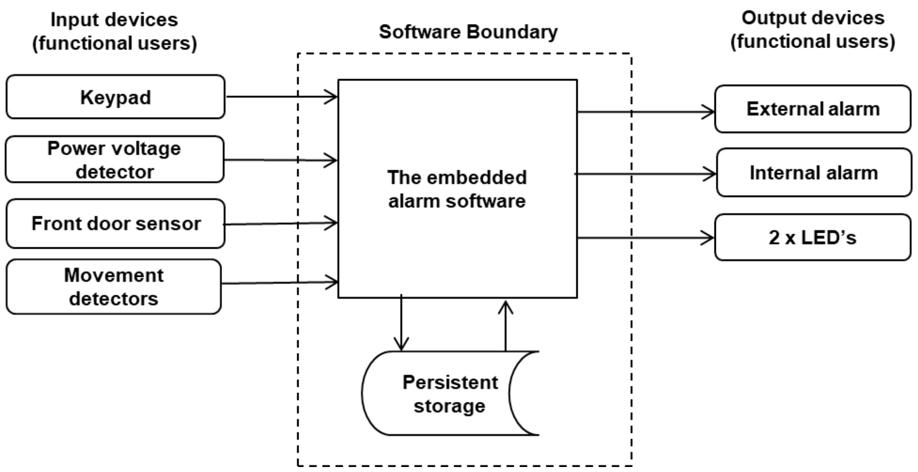


Figure 3.3 – The Intruder Alarm System Context Diagram

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
**	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 and the internal siren earlier in the process, but with different data values ('on' instead of 'off', and vice versa).

The total size of this functional process is 8 CFP

3.5 Cooker software

A 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 is in a ‘standby’ state. 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. Context diagram

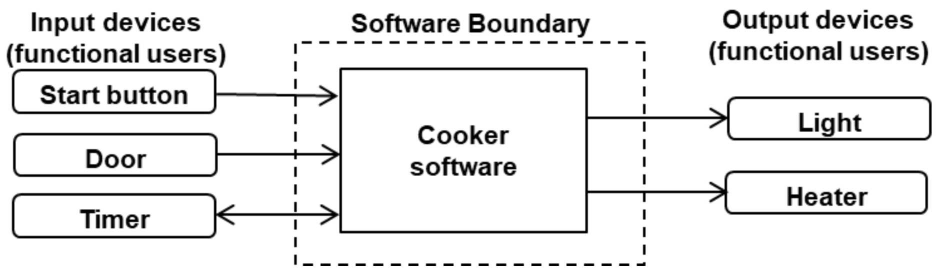


Figure 3.4 – Cooker Context Diagram

The state transition diagram of the cooker is shown in Figure 3.5. 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 (occurrences of) triggering events. These are prefixed by ‘TE’ and the functional users that sense the events by 'FU'.

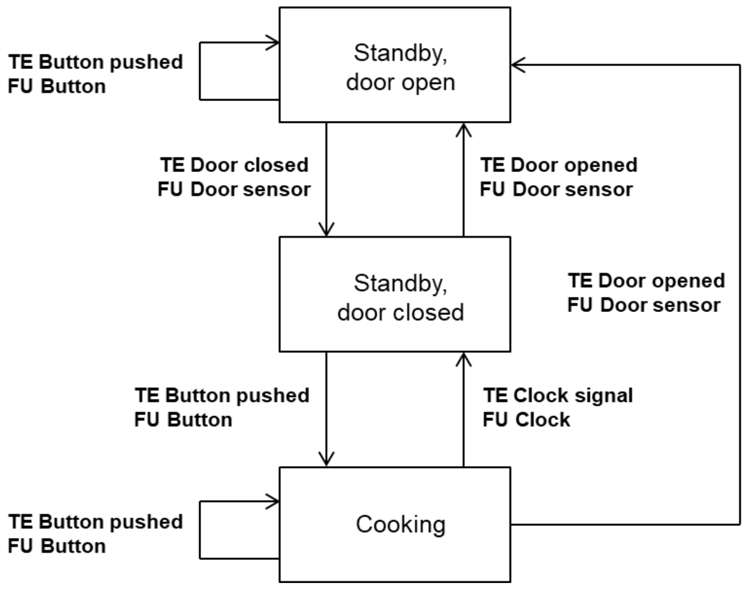


Figure 3.5 – Cooker State Transition Diagram

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. 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 3.2 - 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 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.

3.6 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

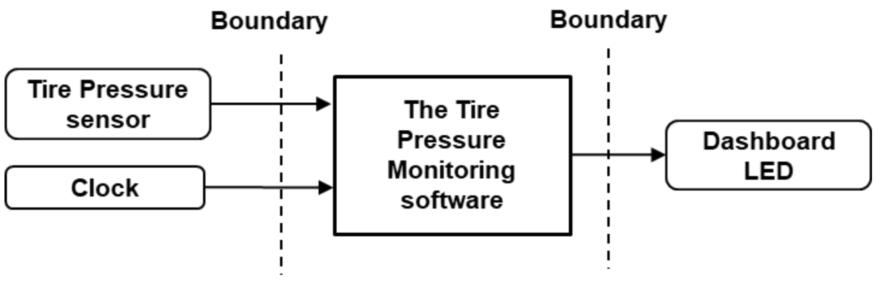


Figure 3.6 – TPMS, context diagram

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 3.3 - 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.

3.7 Automation of sizing real-time requirements

EXAMPLE. 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] and n [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].

3.8 Measurement of data manipulation-rich real-time software

EXAMPLE 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]. 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 3.7 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.)

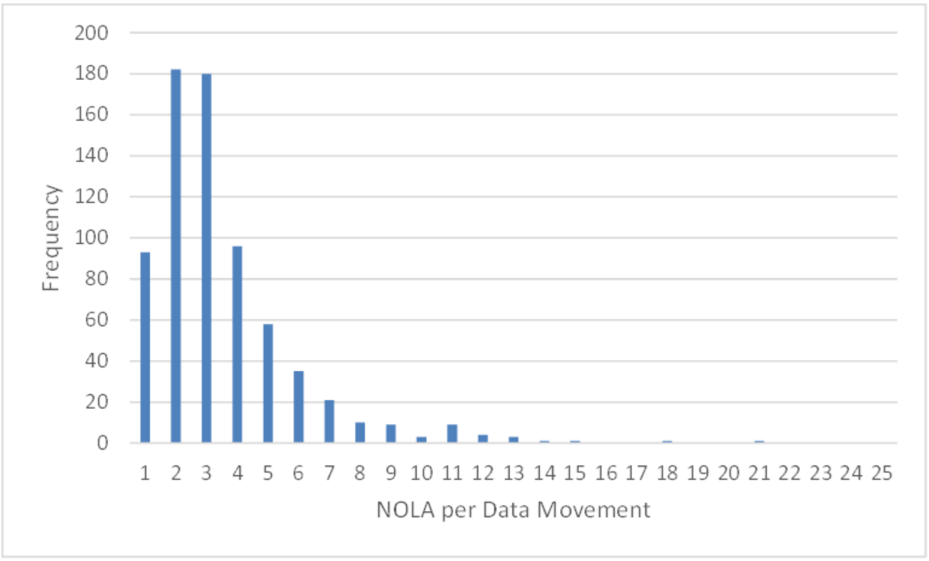


Figure 3.7 - Frequency of NOLA per Data Movement

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.

3.9 Sizing the memory requirements of vehicle Electronic Control Units

EXAMPLE. The study in [12] 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.

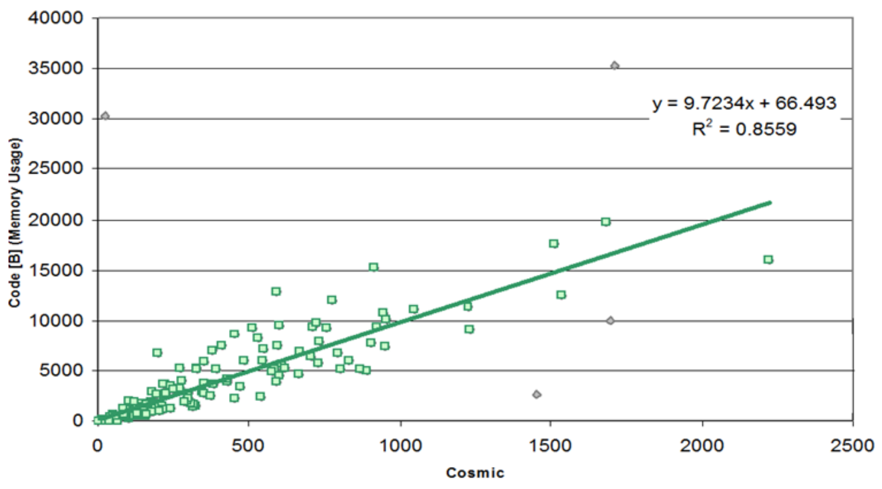


Figure 3.8 - Code size (bytes) versus COSMIC functional size (CFP) [12]

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, v5.0 Part 1, Part 2, Part 3 (*)

[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 Requirements Practitioner / Expert (*)

[6] COSMIC, Early Sizing Practitioner / Expert Guides (*)

[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’, International Workshop on Software Measurement - 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 Renault 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, International Workshop on Software Measurement – IWSM-MENSURA , Stuttgart (Germany), November 2010 - IWSM 2010 (*)

[13] Gencel, C., Heldal, R., and Lind, K., ‘On the Conversion between the Sizes of Software Products in the Life Cycle’, International Workshop on Software Measurement – IWSM-MENSURA, Stuttgart, November 2010.