COM200

Ventilator Software

Name: Omar Darweesh

Student Number: 20173517

Department: Computer Engineering

Maquet Ventilator software system is a distributed real-time system. A bunch of PC boards are carried on the ventilator. Each PC board has its own CPU to coordinate all the on-board electronics and all the boards can communicate with each other and keep synchronized. In this way, the PC boards carry diverse functions and play different roles in the entire system. They cooperate with each other and meanwhile get rid of interferences from each other. Figure 3.1 profiles the architecture of a Maquet ventilator. As we can see, the ventilator system consists of three primary PC boards: Breathing subsystem, Monitoring subsystem, Panel subsystem. Appendix A gives a more detailed system main block diagram. Each individual card is an embedded Linux system with its own ARM processor, memory and peripheral I/O interfaces. Specific software subsystems are installed on the corresponding embedded system.

Identifying plenty of advantages of distributed system development. Combined with the Maquet ventilator system, we could summarize as the following:

* Resource Sharing A distributed system allows the sharing of hardware resources. The straightforward example here are the sensors shared by breathing and monitoring subsystems.
* Parallel Computing In distributed systems, several processes could operate at the same time on separate computers on the network. Different subsystems run on separate computers
* Fault tolerance the availability of several computers and the potential for replicating information means that distributed systems can be tolerant of some hardware and software failures.

**The Breathing Subsystem**

The Breathing Subsystem controls the ventilation system, which is the ventilation control and regulation, inspiratory channel and expiratory channel. Electronics including microprocessor handle of

* Respiratory timing pattern including respiratory ratio as well as distribution of the duration for inspiration, pause time and expiration time according to the front settings.
* Regulation of inspiratory flow and pressure during inspiration time. The desired total inspiratory flow value according to front panel settings is used to generate the flow reference signals for both oxygen and air supply flows. The level relation between these two flow reference signals depends on the desired O2 concentration according to front panel setting. These two reference signals are used for the control of its respective Gas Module
* Regulation of expiratory flow

If we model the medical ventilator as an embedded control system, the breathing subsystem is the controller of the system. The breathing subsystem keeps observing gas chain characteristics, including all the breathing parameters. The difference between the actual measured value and the target value would be taken as input of the breathing subsystem. Besides, the breathing subsystem will listen on some system logic signals as well, taking the power condition and alarm buzzer as examples. After some advanced calculation, which depends on the front-panel setting mode and arguments, breathing subsystem would export some actuator reference signals to regulate the inlet and outlet flow and pressure. The input signal set consists of 24 analogue breathing parameters and 24 system logic signals in total. Meanwhile, there are 4 analogue reference signals and 24 logic signals which are exported to gas chain electronics. The table lists some of the input signals that we are interested in. As known to all, micro-controller could only process digital data while the sensors output some analogue voltage. And it would be time-consuming for the micro-controller to poll all the ADC channels. This explains the existence of the FPGA between the gas chain and the microprocessors in Figure below. FPGA is presented to bridge analogue and digital world. FPGA, rather than the CPU directly, would poll the ADC (analogue-to-digital converter) channels periodically at 2000 Hz. All the ADC signals would be stored on a block RAM. At the same time, FPGA applies the actuator reference value to the

|  |  |  |
| --- | --- | --- |
| Signal Name | Signal Type | Subsystem (Mon/Bre/Both) |
| Expiratory Pressure | ADC signal | Both |
| Oxygen Flow | ADC signal | Both |
| Barometer Pressure | ADC signal | Both |
| Safety Valve | Digital Input | Both |
| Buzzer | Digital Input | Mon |
| Disable Valve | Digital Input | Mon |

valves through the DAC (digital-to-analogue converter) channels. On the other side, the FPGA block RAM would be synchronized with the CPU on the SPI bus. Instead of sampling the gas chain directly, CPU would communicate with FPGA and read data from the block RAM which is very efficient. And it is the same story for the monitoring subsystem

**Monitoring Subsystem**

As discussed, medical ventilators are highly safety critical system. Even with the most careful fault avoidance development, faults will eventually occur and result in a system failure. Fault-tolerance techniques is mandatory in the medical ventilator development. Fault-tolerant system development should be based on a dependable process. A dependable system architecture is also essential for dependability. Ian Sommerville explains that such a system architecture should be designed to include redundant components and mechanisms that allow control to be switched from one component to another. Laprie classified redundancy into four types: hardware redundancy, time redundancy, information redundancy and software redundancy. The simplest realization could be a replicated server, where two or more servers carry out the same task. Replicated servers provide redundancy but not usually diversity. Diversity means that the redundant subsystems are of different types, different functionality, thus increasing the chances that they will not fail in exactly the same fault

Sensors



Sensors

Protection

System

Controller

System

Controllled System

Actuators

System Environment

Sommerville summarizes a protection system with both redundancy and diversity. Figure above. illustrates the relationship between the protection system and the controller system. Both the controlled system and environment are monitored by the protection system to provide improved reliability. Maquet Ventilator adopts such a dependable system architecture and the Monitoring Subsystem is the protection subsystem here. The protection subsystem only includes the critical functionality that is required to move the system from a potentially unsafe state to a safe state. As the name implies, the monitoring subsystem controls all monitoring and alarm functions in the system, including trends of measured values. Events, such as alarms and change of system settings would be logged. If some problems detected, it issues alarms and leads the system to a safe state, which means to recover from the failure or just to shut down and stop service. To guarantee the patient one hundred percent safe, it activates pressure reducing mechanisms, including activation of the safety valve, in case of excessive breathing system pressure. All alarms are conveyed and displayed on the front panel and the alarm sound is also generated. In case of malfunction in the loudspeaker located on panel, a back-up sound generating device (buzzer) on Monitoring PC board will be activated automatically. This buzzer is monitored by a microphone at start-up and during the pre-use check. The buzzer on monitoring card generates the alarm signal in case of +5 V or +3.3 V power failure. The buzzer and +5/+3.3 V failure logic is powered by back-up capacitors in case of power failure.

**Panel Subsystem**

The panel subsystem implements all the user ventilator interaction, as well as software updating to all subsystems via the PC card interface. User could configure patient information and technical parameters on the touch screen.



What's more important, real time breathing parameters could also be displayed on the panel. During ventilation, measured or calculated values of breathing parameters are displayed. Besides the breathing parameters, some color-coded waveforms will be shown on the user interface screen as well. By default, they are:

* Pressure vs Time
* Flow vs Time
* Volume vs Time

Thus, the user could have a straightforward knowledge of the patient's respiratory condition.

Foremost, the warning and alarms will be displayed on the user interface screen.

**Interaction and Communication**

Distributed systems are more complex than systems that run on a single processor. There are several important issues that have to be considered in distributed systems engineering. One of the most significant things is Communication and Interaction. It includes the inter-subsystem communication and the inner communication between different components in a single subsystem. Today the principal techniques applied in Maquet ventilator are Serial Peripheral Interface (SPI) and Google Protocol Buffer (GPB).

**Serial Peripheral Interface**

SPI bus is designed to exchange data between CPU and FPGA block RAM in both breathing and monitoring subsystems. It is also utilized between breathing and monitoring subsystems. The SPI bus is short for Serial Peripheral Interface. It's a four-wire serial bus and devices work in master/slave mode, were the bus master always initiates the message transaction. The slaves are allowed by individual chip select lines.

**Google Protocol Buffer**

Google protocol buffer is a Language neutral, platform neutral and extensible way of serializing structured data. Protocol buffers are developed, used and well documented by Google. It is flexible, efficient and automated, like XML, but smaller, faster and simpler. Another advantage of protocol buffer is that it supports multiple languages, Java, C++ and python as well. Protocol buffers are now Google's lingua franca for data. They're used both in Remote Procedure Call (RPC) systems and for persistent storage of data in a variety of storage systems. You can specify your own data structure and details of the information being serialized. Each protocol buffer message is a small logical record of information, containing a series of name-value pairs. Then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages. There are three steps to define a nice tidy protocol buffer message.

1. Specify Field Types Data in messages is hierarchically structured. Each message has one or more uniquely numbered fields. Each field has its name and value. Google protocol buffer supports various value types. It can be numbers, no matter integer or floating point. It can be Boolean, strings and raw bytes. What's more, you can even encapsulate other protocol buffer message types, which makes it more flexible and powerful.

2. Assign Unique Tag As mentioned, each field in the message definition has a unique numbered tag. These tags are used to identify your fields in the message binary format, and should not be changed once your message type is in use. Note that tags with values in the range 1 through 15 take one byte to encode, including the identifying number and the field's type. Tags in the range 16 through 2047 take two bytes. So, you should reserve the tags 1 through 15 for very frequently occurring message elements. Remember to leave some room for frequently occurring elements that might be added in the future.

3. Specify Field Rules All the message field should be specified as one of the following:

* **Required** a well-formed message must have exactly one of this field
* **Optional** a well-formed message can have zero or one of this field (but not more than one)
* **Repeated** this field can be repeated any number of times (including zero) in a well-formed message.

The order of the repeated values will be preserved. Today, protocol buffers are applied in Maquet ventilator systems as well. Messages goes through Ethernet and synchronize different subsystems.

**Software Testing**

As a highly safety critical system, it is essential that the applied software is trustworthy.

The European Standard EN 60601-1:2006 has defined the basic safety requirement

as freedom from unacceptable risk directly caused by physical hazards when medical

electrical equipment is used under normal condition and single fault condition.

The software should be available to provide service when required and should operate

correctly and without undesired risks and hazards

**Dependability and Fault tolerance**

The term dependability was proposed by Laprie in 1985 to cover the related system attributes of availability and reliability. As Laprie delivered in his article, achieving a dependable computing system calls for the combined utilization of fault-avoidance, fault-tolerance, error-removal, and error-forecasting, where fault-avoidance and fault-tolerance may be seen as constituting dependability procurement. Fault-tolerance tightly relates to safety critical systems that should include facilities to handle with abnormal or anomalous conditions. All the software system is designed to provide specified service. System dependability is the quality of the delivered service such that reliance can justifiably be placed on this service. If the system stops delivering the intended service, we call this a failure.

We call the cause of failures fault. A fault causes an error in the internal state of the system and the error eventually results in the system failure. Faults can be hardware faults or software faults. Component defect is, of course, a hardware fault. For example, a pressure sensor in the gas chain fails to measure the pressure correctly. Hardware fault could also be a circuit break. Software fault is commonly caused by code implementation. But a fault can also be the external environment and disturbances like electromagnetic interference and operator mistakes.

The great diversity of faults makes the evaluation of a fault-tolerant system a definitely complex task

A great deal of studies, taking, have shown

the prominent efficiency of the fault tolerance algorithms and mechanisms (FTAM's)

on the dependability of abundant systems and architectures. Among the possible approaches

such as proving or analytical modelling, fault injection is proven to be more

attractive.

The main idea of fault injection is to speed up the occurrence of errors and failures. Itis a method for testing the FTAM's with respect to their own specific faults, that they

are expected to stand. points out that fault injection addresses both error removal

and error forecasting. With respect to the error removal objectives, fault injection is

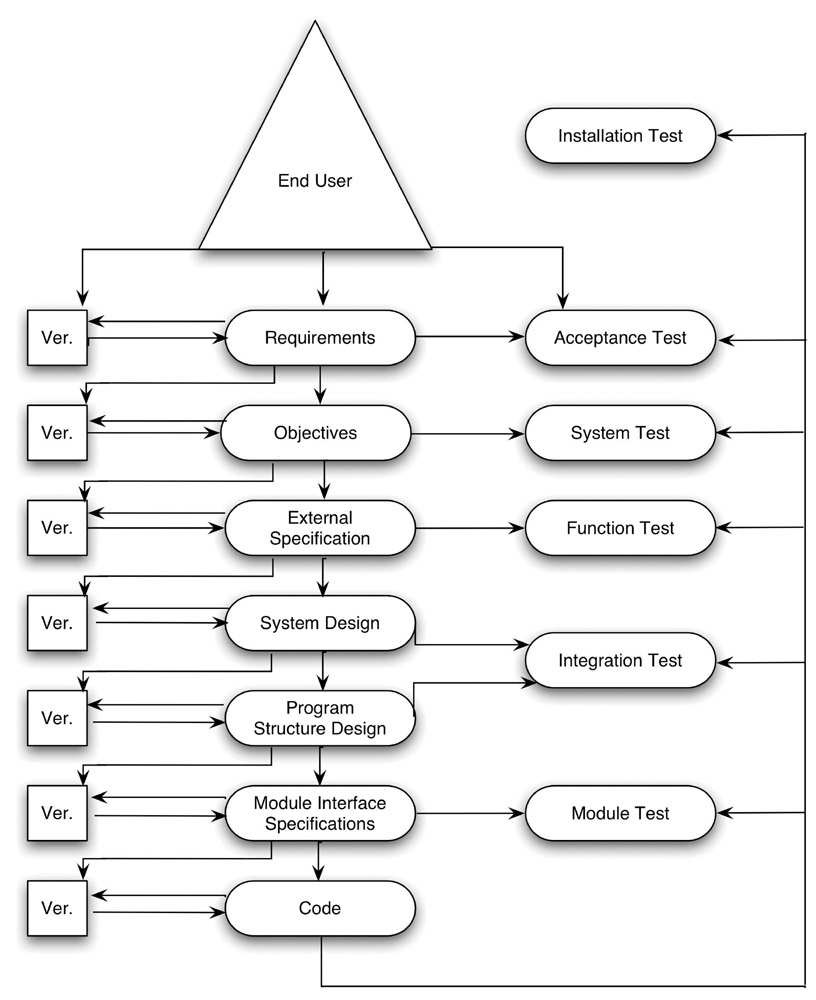
explicitly target at reducing, by verification, the presence of faults.

**Formal Testing**

Software testing is, as Myers defines in his book. the process of executing a program with the intent of finding errors. He divides testing before release into four main stages:

* Module (Unit) testing
* Integration testing
* Function testing
* System testing

The next figure illustrates the testing cycle structured to model the development cycle. Testing is first focused on the small individual components of the whole system, starting from module testing. After verifying the unit functional and interface specification, developers move to integration testing. As the name implies, integration testing works to expose the potential defects in the interaction of different modules. Progressively, larger groups of modules and components could be integrated and tested until systems. Both module and integration testing are conducted by developers during development phase. As indicated in the Figure below, function testing is a process of attempting to dig out the deviations between the application and the external specification. An external specification is a precise description of the system’s behavior from the point view of the end user System testing is the most confusable process. It’s not a process of testing the functions of the complete system. Instead, system testing is to compare the system to its original objectives. It’s impossible if there is no set of written, measurable objectives for the system. Professional testers should take over the function and system testing instead of developers themselves. Usually, black-box testing is applied in this stage, because the internal logic and structure is unconcerned, which means its input/output driven strategy.



**Challenges at a Maquet**

Today at Maquet, we could classify software testing into two categories: normal condition testing and failure mode testing. Normal condition means condition in which all means provided for protection against hazards are intact and operations according to the instructions for use. All the normal mode formal testing has already been automated and it’s proven to be prominently efficient and trustworthy. However, it’s still difficult to realize test automation under failure mode. Failure mode means that, for some uncertain faults, the system fails eventually.

The software system should somehow recover from this abnormal and anomalous conditions and continue providing service or just shut down and stop service, which is safety critical system supposed to do. Bluntly, tremendous manual testing are ongoing at Maquet in failure mode testing even today.

We can first have a look at an example. A barometric pressure too low alarm shall be triggered if the measured barometric pressure is below (650±10) hPa(refer to Appendix B) and the barometer pressure used internally in the Monitoring Subsystem shall be limited to 650 hPa if a lower pressure is detected.

In this example, the error is the measured barometric pressure. This could result from the barometer defect or maybe because of circuit break., software system read breathing parameters from the FPGA block RAM and FPGA polls the sensors and updates the block RAM at a high frequency.

Which means that a single error would be flushed and overwritten in the next sampling cycle. The tight relation between the software system and the hardware components makes it much more challenging to inject the expected fault.

Before this thesis, manual job is mandatory to simulate and inject some barometer error. Testers have to apply a certain voltage that corresponds to the barometric pressure to the output pin of the barometer. This manual job counts up to 6 times in one single test case. (refer to Appendix B).

This is an obvious obstacle to test automation. The importance to test automation has been recognized long ago and is undisputed today. manual testing is time consuming and cost consuming from business concern.

What’s worse, manual testing is error prone. Manual job is always potential fault source and the testing quality is not guaranteed. Rather than manual testing, automated testing has convinced to be much more promising. Alongside noticeable efficiency, automated testing has more controllability and reproducibility.

To summarize, the vast quantity of hardware dependencies becomes the bottleneck of automated software testing, especially in failure mode. Because manual fault injection cannot be replaced by automated implementation today

**Integration with AVA**

AVA is a test framework that simplifies and unifies the procedure of writing advanced system and subsystem tests. The purpose of AVA is to allow for automated regression testing, reduce maintenance work, and to help test writers to understand each other’s test code. AVA is written in Python and runs primarily on Linux systems, mainly because the original test targets (BRE/MON/PAN) are built as Linux ELF binaries. AVA uses the Python unit-test framework for its test cases. The unit-test framework gives deep control of the test flow, and provides functions for separating test methods, setup code before each test method, tear-down code after each test method, pre-test setup code, etcetera. A test case is a collection of test methods needed to test a specific requirement. The test method setup is predefined in unit-test and is used to test various aspects of for example a requirement. A test case may contain one or several test methods. When a test case is executed, AVA will run all test methods defined in the test case, gather statistics and present a result for the test case.

**Comparison and Conclusion**

In conventional testing, manual maneuvers is mandatory due to the tight hardware dependencies. If there is only a small number of test cases, manual testing is acceptable. It requires little time and expense to begin productive manual testing and short-term costs are reduced.

However, manual testing could be very time consuming when there are plentiful test cases. Here we just give one single test case. However, it is only the tip of the iceberg and there are a vast amount of test cases which require manual maneuvers.

Considering the fact that you must rerun the same set of test cases for each single release, it is tiresome. What’s worse, it is error prone. Taking the first step as an example, the integrated circuit is of great complexity and the pins are intensively crowded together. Testers might apply the voltage to a wrong pin.

Various faults might be introduced by manual job. Therefore, the quality of manual testing is not guaranteed. The presence of the embedded patient simulator makes test automation possible, which is a momentous step for verification methodologies at Maquet.

The embedded patient simulator works as a standalone module during software testing. the gas chain dependency is removed and instead, the simulator is interfaced with the ventilator.

The simulator reads the actuator reference values and returns sensor value for each breath moment. The ventilator software system runs on target which means that all the real-time properties of the distributed real-time system could be tested.