

Development of a Framework Useful in Creating Virtual Panels for PFBR Operator Training Simulator

Ganesh Patel

ENGG01201601042

Bhabha Atomic Research Centre, Mumbai

*A thesis submitted to the
Board of Studies in Engineering Sciences
In partial fulfilment of requirements
for the Degree of*

MASTER OF TECHNOLOGY

of

HOMI BHABHA NATIONAL INSTITUTE

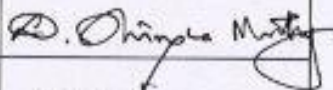
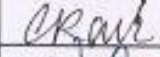
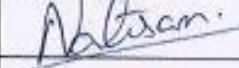
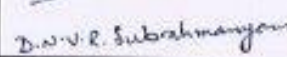
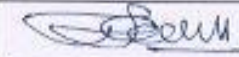
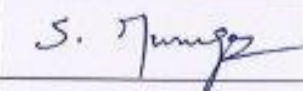


January, 2019

Homi Bhabha National Institute

Recommendations of the Thesis Examining Committee

As members of the Thesis examining Committee, we recommend that the thesis prepared by **Ganesh Patel** entitled "**Development of a framework useful in creating virtual panels for PFBR operator training simulator**" be accepted as fulfilling the thesis requirement for the degree of Master of Technology.

	Name	Signature
Member 1	Dr. D. Thirugana Murthy	
Member 2	Dr. Ravi Chinnappan	
Member 3	Mr. K. Natesan	
Technology Advisor	Mr. DNVR Subrahmanyam	
Examiner	Dr. D. Thirugana Murthy	
Guide & Convener	Dr. M. L. Jayalal	
Chairman	Dr. S. Murugan	

Final approval and acceptance of this thesis is contingent upon the candidate's submission of the final copies of the thesis to HBNI.

I hereby certify that I have read this thesis prepared under my direction and recommend that it may be accepted as fulfilling the thesis requirement.

Date: 30/01/2019

Place: Kalpakkam



Dr. M. L. Jayalal

DECLARATION

I, hereby declare that the investigation presented in the thesis has been carried out by me.
The work is original and has not been submitted earlier as a whole or in part for a degree /
diploma at this or any other Institution / University.

Ganesh
30/01/2019
Ganesh Patel

*To my family, friends, and all those
who work for Fast Breeder Reactors*

Acknowledgements

I was extremely grateful to have Dr. M. L. Jayalal as my guide and thesis advisor. His encouragements and continuous motivation kept me going. It would not have been possible to complete this work without his constant effort and guidance. I express my gratitude towards him for all the things I learnt under the environment provided by him. Along with him my Technology advisor Shri DNVR Subrahmanyam was a constant supporter who always listened to what I have to say and he was always ready with suggestions. We used to have endless discussions on approaches to address a problem and I thank him for the effort he took to explain me all the intricacies involved in developing the software. He has provided a great deal of knowledge about simulator in such short period of one year. He introduced me to the project and suggested to take up this work as project for my post-graduation. I would also like to thank Shri. R. Jehadeesan, who provided the environment to work and completing all the official formalities regarding the security and project.

I would like to thank my reporting officers Shri S. Narasimhan, ACE(IT & Instrumentation), BHAVINI and Smt. H. Nalini, BHAVINI who both always helped me to focus on my project work and job synchronously. Any kind of help I ask for, they will always be ready with the helping hand. Along with them I would like to thank Director, Resources and Documentation Division, Atomic Energy Regulatory Board (AERB) and Shri S. A. Bhardwaj, Chairman, Atomic Energy Regulatory Board (AERB) who encouraged me for the project work.

I would also like to thank my committee members for their support, valuable and constructive comments which shaped the thesis and project work in the appropriate way. In addition to them HBNI as an institution has played a very important role in shaping the guidelines for the coursework and project work. I would like thank all the HBNI apex committee and staff who are constantly working to provide this opportunity for the students.

My big thanks to my friends, Vaibhav Khulbey, Nishu, and Ritu Rani for making life happier and easier. Thanks to everyone who tolerated me during the tenure of the work, as at times circumstances demanded to be present at many places simultaneously.

Finally, I would like to thank my family for their love and encouragement.

Contents

Synopsis.....	I
List of Figures.....	III
List of Algorithms	V
List of Pseudocodes	VI
List of Tables	VII
1 Introduction.....	1
1.2 Simulator for Nuclear Reactors	2
1.2.1 Types of Simulator	4
1.3 PFBR Simulator	5
1.3.1 Simulator Specifications.....	5
1.3.1.1 Plant consoles.....	5
1.3.1.2 Computer System.....	5
1.3.1.3 Instructor Station.....	6
1.3.2 Limitations of PFBR Simulator.....	6
1.3.3 Solution Proposed for overcoming the limitation of PFBR simulator	8
1.4 Scope of the Project.....	8
1.5 Thesis Outline	9
2 Hardware and Software Architecture of the Simulator.....	10
2.1 Hardware Architecture of PFBR Simulator	10
2.1.1 Simulation Computer.....	11
2.1.2 Control Room - Panels and Consoles	11
2.1.3 I/O System	11
2.1.4 Local Control Centre (LCC).....	12
2.1.5 Data Highway	12

2.1.6	Instructor Station	13
2.1.7	Process Computers	13
2.2	Software Architecture of PFBR Simulator.....	13
2.2.1	Daemon Processes	15
2.2.1.1	Database Server	16
2.2.1.2	MDSM	16
2.2.1.2.1	Callback Function	17
2.2.1.2.2	Registering a Process	18
2.2.1.2.3	Registering and De- Registering the Variables	19
2.2.1.2.3.1	Registering for Publishing	19
2.2.1.2.3.2	Registering for Subscription	20
2.2.1.2.3.3	De-Registering the Variables.....	22
2.2.1.2.4	Publishing Data	23
2.2.1.2.5	Fetching Data	24
2.2.1.2.6	Sending a Message	24
2.2.1.2.7	Receiving a Message	25
2.2.1.2.8	De-Registering the Process	26
2.2.1.3	Logger	26
2.2.2	Process Models	27
3	Interaction between Models of the Simulator	29
3.1	External Models	29
3.2	Basic Operation of Simulator Component	31
3.2.1	Initialisation	31
3.2.2	Run Time	32
3.3	Interaction between Simulator Components	33
4	Development of Framework to Create Virtual Panels	35
4.1	Virtual Panels	35
4.2	Tools used for Development	36

4.2.1 Qt	36
4.3.2 SQLite.....	37
4.3 Basis for selection of tools of development	38
4.4 Types of Signals	39
4.5 Development Methodology	40
4.6 Details of Data Structures Used	41
4.7 Development of Panels.....	47
4.8 Panel Components and their Implementation	49
4.8.2 Pushbutton	50
4.8.2.1 Momentary Type Pushbutton.....	50
4.8.2.2 Latching Type Pushbutton	51
4.8.3 Display Component	52
4.8.3.1 LED	52
4.8.3.2 Semaphore Indicator	53
4.8.3.3 Digit Display	54
4.8.3.4 Bar Graph Indicators	55
4.8.4 Selector Switches.....	55
4.8.4.1 Normal Selector Switches.....	56
4.8.4.2 Key Selector Switches	56
4.8.4.3 Gang Selector Switches	58
4.9 Parsing the User Interface (UI) and Storing Information.....	58
4.10 Pre-processing of the Information and Registering the Variables	63
4.11 Interaction with User Interface (UI).....	68
4.11.1 Pushbutton	68
4.11.2 Selector Switch.....	69
4.11.3 Toggle Switch.....	70
4.12 Updating the User Interface (UI)	71
4.13 Results of the Development	74

5 Integration of Virtual Panels with the Simulator	78
5.1 IPC Implementation and Registering Process with Simulator	78
5.2 Communication with the Executive	81
5.3 Sending User Interface (UI) Responses to Simulator	82
5.4 Saving the state of User Interface (UI) and De-Registering the Process and Variables	82
5.5 Results of Integration	83
 6 The Methodology of reusing the Framework.....	86
6.1 Creating a virtual panel	86
6.2 Incorporating Components in the Virtual Panel.....	87
 7 Summary and Future Work	89
7.1 Future Work	90
 Annexure A.....	91
A.1 Database Files used for DB3 setup	91
A.2 Types of Widget Component and Actions with Codes	92
 Annexure B.....	93
B.1 Pseudocode for the Algorithms	93
B.1.1 Pseudocode for Algorithm 4.1	93
B.1.2 Pseudocode for Algorithm 4.2	95
B.1.3 Pseudocode for Algorithm 4.3	95
 References.....	98

Synopsis

Nuclear power plants require highly skilled and trained manpower in order to reduce the possibility of any undesirable event. Training simulators are a positive step towards safety of nuclear power plants. Prototype Fast Breeder Reactor is a nuclear reactor situated at Kalpakkam, India. An Operator Training Simulator (OPTS) is designed and developed for training the operators of PFBR. PFBR simulator is a full scope replica type simulator. The simulation computer is heart of the simulator as it performs all the calculations and computations based on the mathematical models of the physical systems of the plant. PFBR simulation computer is based on ALPHA architecture which is obsolete now. The Tru64 Unix operating system is fully hardware dependent and in case of failure of processor, operating system as well as simulation software would stop working. The approach followed to overcome this problem was to port the simulator to Intel platform with an open source operating system which is hardware independent. Simulator consists of three main models - process models, logic models, and virtual panels. Among these models, process models and logic models are already ported. This project focuses on development of framework to create virtual panels using an open source tool and integrating them with the other two models. Testing of the framework is done by creating virtual panels for some hardware panels and integrating them.

Virtual panels are the soft panels which emulate hardware control panels with animated equipment widgets to virtualize operator interface. Virtual panels are useful during commissioning phase and during failure of hardware panels. Widgets show real time

behaviour for changes in application data. These include alarms, pushbuttons (momentary and latching type), selector switches, gang-selector switches, LEDs, display segments like LCD, bar indicators, ammeters, voltmeters, synchroscope, etc.

Tools used for development are Qt and SQLite. Both of these tools are open source under General Public User's License (GPL). Development of a generic framework has been carried for creating virtual panels and addition of new components to the project. Framework enables addition of new virtual panels and components to the project thereby reducing development time and effort for expansion.

The framework is developed in such way that virtual panel is parsed and information related to components is extracted and stored into a database. Not saving the information in database will require parsing the virtual panels multiple times, which would degrade the response time of the panels. Data is accessed for the virtual panel currently open and not for all the panels. Databases used in the project are in-memory databases which are much faster than the disk storage databases.

List of Figures

1.1	Flow of development of simulator.....	4
1.2	Simulation Computer.....	7
2.1	I/O system and its interactions.....	12
2.2	Organisation of simulator software components.....	14
3.1	Initialisation of simulator components.....	31
3.2	Runtime operation of simulator component.....	32
3.3	Simulator software system and its components.....	34
4.1	Top Screen of virtual panels.....	47
4.2	Alarm tab of panel 1.....	48
4.3	(a) State of no alarm for Log No. (b) State of alarm for PCSL LOGIC HEALTHY	49
4.4	(a) ALARM TEST pushbutton in released condition. (b) ALARM TEST pushbutton in pressed condition.....	51
4.5	(a) EMERGENCY pushbutton in released condition. (b) EMERGENCY pushbutton in pressed condition.....	51
4.6	(a) RED color LED. (b) GREEN color LED. (c) YELLOW color LED.....	52
4.7	(a)Two LED set for each of the three DSRDM(ENERGISED/DEENERGISED) (b)Two LED set for each of three DSRDM(LOCKED/UNLOCKED) (c)Two LED set for each of nine CSRDM(ENERGISED/DE-ENERGISED)	53

4.8	(a) Semaphore indicator with value 1	
	(b) Semaphore indicator with value 0.....	54
4.9	Digit display.....	54
4.10	Bar graph indicator.....	55
4.11	(a) Two position Selector Switch(SS). (b) Three Position SS. (c) Four Position SS. (d) Five Position SS. (e) Eight Position SS. (f) Twelve Position SS.....	57
4.12	(a) Two positions key selector switch.....	57
4.13	Key selector switch and two LED combination.....	58
4.14	(a) Gang Selector for DSR selection	
	(b) Gang Selector for range selection.....	58
4.15	ALARM tab of panel 1.....	75
4.16	DISPLAY tab of panel 1.....	76
4.17	CONTROL tab of panel 1.....	77
5.1	Integrated ALARM tab of panel 1.....	83
5.2	Integrated DISPLAY tab of panel 1.....	84
5.3	Integrated CONTROL tab of panel 1.....	85

List of Algorithms

4.1 Identification of component widgets and storage of information.....	59
4.2 Insertion of data into databases.....	63
4.3 Segregation and storage of component widgets.....	65
4.4 Updating virtual panel and component widgets.....	72

List of Pseudocodes

4.1 Actions when pushbutton is pressed/released.....	69
4.2 Actions when selector switch pushbutton is Pressed/Released.....	70
4.3 Actions when toggle switch pushbutton is Pressed/Released.....	71
4.4 Identification of components widgets and storage of information.....	94
4.5 Insertion of data into the databases.....	95
4.6: Segregation and Storage of component widgets.....	95

List of Tables

2.1 MDSM message types.....	18
4.1 Comparison of Qt and GLG toolkit.....	39
4.2 Table for storing widget information.....	42

Chapter 1

Introduction

Computer systems form an integral part of our life. They are being used in various safety critical applications. They are used in simulating the safety critical systems. This chapter describes the need for a simulator for a nuclear power plants, specifically in training of manpower for efficient operation of the reactor. There are various types of simulators based on requirement and type of usage. This chapter also explains the need and objectives of the project and underlying limitations of the PFBR simulator.

1.1 Prototype Fast Breeder Reactor (PFBR)

Prototype Fast Breeder Reactor (PFBR) [1,2] is a 500 MWe pool type, mixed oxide fuelled, sodium cooled fast reactor situated at Kalpakkam, India. The core thermal capacity is 1250 MWt. Design and layout of the reactor has been made taking into consideration safety requirements, constructability, maintainability, security and economics.

No industry is immune from accidents, but all industries learn from them. Safety focuses on unintended conditions or events leading to radiological releases from the authorised activities. Most of the events which are known as design basis events can be

thought of beforehand and actions required during these events can be planned in advance. Training the nuclear reactor operator is one of the most effective ways to avoid the safety related incidents. An operator should not be exposed to any conditions during the operation of reactor which the operator has not encountered earlier. One of the most effective ways to reduce safety related incidents is to expose and train the operator before the event itself. This can be done by simulating the events in a much safer environment like simulator rather than in nuclear reactor itself.

1.2 Simulator for Nuclear Reactors

Nuclear reactors are one of the most critical systems in the world. Need of simulator for a nuclear power plant was envisaged as soon as the reactors became common. Human error is evidently the major reason for accidents as seen in past. Simulators play a vital role in enhancing plant safety by the reducing human errors. However, simulators cannot simulate the mechanical issues such as fuel failure, beyond design basis accidents, etc. Operators of nuclear reactor are first trained on the simulator before starting work in nuclear reactor.

Training[1,3,4] is given to the new operators as well as the qualified operators as a refresher course to maintain their skills at peak levels. Computer based Operator Training Simulator (OPTS) is a vital tool for this purpose. Computer based OPTS plays a major role in imparting best possible training to operators, due to the flexibilities and functionalities it provides. It is possible to impart operation training and qualify the operator even before the reactor is commissioned. OPTS can be initialized to one of the operational states or it can be switched from one operating state to another without any delay. This is a very useful feature for operation training and contributes significantly towards reducing the training time. Most of the malfunctions and emergency drills for which training is required cannot be allowed to

occur in the reactor due to safety and economic reasons. This will cause hindrance to normal operation. OPTS offers this flexibility with ease and elegance.

If a trainee operator makes a mistake while training in real plant, the instructor has to step in and correct the mistake for reasons of safety. It would be more desirable to let the trainee see the consequences of error made, to make the most lasting impression of the proper operating procedures. Features like freezing and backtracking facilitate this in the simulator. For example, operator raises the control rod beyond the required limit, then instructor can step in, freeze the simulator and explain the conditions along with the actions to take. Instructor can backtrack to the case where simulated reactor was in safe condition.

Computer simulation [3,4,5,6,7,8] involves developing mathematical models of sub-systems and components of the plant to derive its characteristics and behaviour without actually constructing the system concerned. A simulator is essentially made up of software programs that are designed to replicate a power plant or a critical system. A power station training simulator is a training tool designed to simulate the steady state and dynamic responses of a power station in real time to operator actions. To build an OPTS, the subsystems and components which contribute towards the plant dynamics and those associated with the operation and control functions of the reactor are identified to start with. Their individual behaviour and interlink among them are mathematically modelled. The mathematical model is then translated into computer code that will eventually run in the simulation computer. Flow of development of simulator is depicted in Figure 1.1.

The simulation software is developed in such a way to take input from operator activated control panels, process it in accordance with the plant model and give the computer generated output on the control panel by actuating appropriate devices or their equivalents. It also supports an instructor station from where a training instructor can interact with the

computer software to give appropriate training lessons to the operator, insert different plant scenarios, monitor the operator response and evaluate his performance.



Figure 1.1: Flow of Development of Simulator

1.2.1 Types of Simulator

Training Simulators are broadly classified based on two parameters namely [1,5,8], extent of the plant to be covered in simulation and fidelity in the replication of the plant control room. Based on the extent of the plant to be covered, the simulators are classified as *full scope* or *subsystem* simulators and based on fidelity in the replication of the plant control room, the simulators are classified as *replica* or *non-replica* simulator [9,10,11].

In *full scope* [12] simulators all the systems are covered in the design of the simulator but *subsystem* simulator comprises of only few subsystems of the plant. The accuracy of the models in both the steady and dynamic situations will be high for the *full scope* simulators. In *replica* type, simulators will have a control room and its panels which are one to one replica of actual plant control room, down to desks, chairs and lights. A built-in advantage of the replica type simulator is its ability to do strict procedural training. As with inplant training, trainee can learn location and function of each instrument and control. In *non-replica* simulators, all the indicators and controls may not be same as the real plant control room.

1.3 PFBR Simulator

PFBR Operator Training Simulator is a full scope, replica type simulator with objective of providing comprehensive training to the operators in the nuclear power plant operations. Simulation software models subsystems and components of the plant like neutronics, reactor core, control rod drive mechanisms, core monitoring, reactor protection systems, primary sodium system, secondary sodium system, and steam water system.

1.3.1 Simulator Specifications

The training simulator consists of plant consoles, computer systems and instructor station.

1.3.1.1 Plant consoles

Plant console of the simulator is identical to the operator console and control room panels of the reactor. Operator's console handles overall monitoring and most frequently used controls of the plant. Control room panels are provided on the system basis in order of startup logic, shutdown systems, primary sodium system, secondary sodium circuits, steam generators, turbo generators, electrical power supply, decay heat removal, radiation monitoring, and RCB isolation. The display, indicators and switches in the simulator's plant console are identical to the actual plant console.

1.3.1.2 Computer System

The computer system consists of simulation computer and front end Input/ Output (I/O) computer. The simulation computer computes the plant dynamics and the front end computer handles input and output signals from/to operator console and control room panels.

1.3.1.3 Instructor Station

The instructor station consists of a CRT display, keyboard and mouse. The instructor's console provides for the selection of the mode of operation of the simulator, malfunction selection and monitoring the plant variables. It also provides controls for the initialisation and monitoring of simulator runs. The instructor can pause at any stage of the simulation and continue or back track up to a number of time units. The instructor station also includes recording facility for audio and video to monitor the operator's response including trainee's reflexes.

1.3.2 Limitations of PFBR Simulator

Simulation computer is heart of the simulator system. It executes numerous codes representing the mathematical models of the various sub systems of the plant in sequence and time frame representing the complete behaviour of plant. It accepts various commands from the operator like increase/decrease the flow, raise/lower a control rod, start/stop a pump, etc., processes it accordingly with the mathematical models, generates the responses, plant parameters and outputs them in the required form on the panels and consoles. It updates the database in the plant computer for providing the information in display stations. In order to generate these responses in the real time according to the plant dynamics, the simulation computer should be a powerful system to execute the various mathematical models in time.

Non availability of simulation computer leads to stalling of all the activities of whole simulator system. Simulation computer consists of:

- Compaq Alpha server ES45 with quad alpha chip processors
- Compaq Tru64 Unix operating system

- Cache memory, RAM, Hard disk and other utilities are in compliant with the requirement.

The architecture of the simulation computer is shown in Figure 1.2.

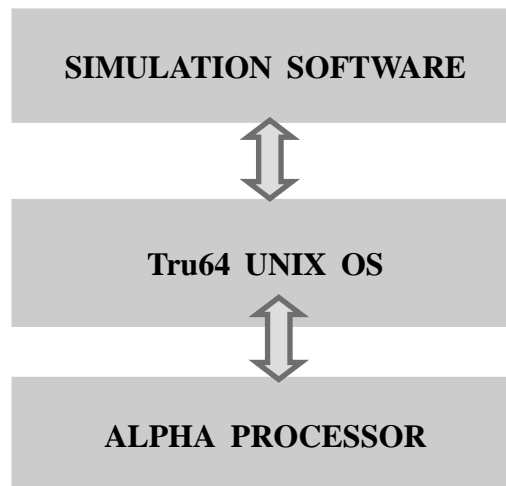


Figure 1.2: Simulation computer

Alpha Server [13] was produced by Digital Equipment Corporation (DEC) which was acquired by Compaq in 1998, which subsequently merged with Hewlett-Packard (1998). Compaq, being an Intel customer sold all intellectual property rights to Intel. HP also discontinued producing more alpha server and discontinued its services in 2008. Compaq also developed operating systems, specifically for alpha servers which includes *Tru64 Unix*. *Tru64 Unix* runs only on alpha based processors. After discontinuation of services for alpha architecture, it is difficult to find spares for alpha server. In case of major problem, revival of the system will be very difficult.

1.3.3 Solution Proposed for overcoming the limitation of PFBR simulator

The approach followed is to use Intel based simulation computer. Intel, being a leader in market will stay longer, and operating system can be processor independent. Even Intel goes out of the market whole code can be again compiled and used for the new processor and operating system chosen, if the operating system exists for the new processor. Operating system chosen is *CentOS* [20], which is a Linux distribution that provides a free, community supported computing platform facility for desktops, mainframes, servers and workstations.

Porting the simulation software to the new platform has reached mature level. Most of the models of the simulator are ported to the Intel platform. For completeness, virtual panels need to be ported. This project deals with the development of framework for creating virtual panels and integrating them with the Intel platform. Development ideas have been taken from the earlier implementation with functional changes to suit the simulator environment. Development is done using open source [21,22] tools Qt and SQLite. Integration of the virtual panels with the other models is also done and tested by developing some of the panels. Development of a framework is done to ease the further development.

1.4 Scope of the Project

Scope of the project covers development of a framework useful for creating virtual panels for PFBR operator training simulator. Virtual panels developed are integrated with the process model and logic model through the framework. Some of the panels are developed to test the framework and their integration with the simulator. Scope of the project does not cover creation of all the virtual panels for the simulator.

1.5 Thesis Outline

Chapter 2 covers hardware and software architecture of the simulator. In hardware architecture, major hardware components and their interactions with the other systems is covered. Software architecture explains the important processes involved in the simulation software. Important processes like daemon processes, process model and logic process are discussed. Major functions of these processes and their interaction mechanism with the other processes is also discussed in detail.

Chapter 3 covers the interaction between the modules of the simulator. It covers addition of new modules to the simulator which are known as external modules. Basic operation of simulator components during initialisation and run-time is also discussed in detail.

Chapter 4 covers development of framework for creating the virtual panels. It discusses the methodology used. Information and data stored in the database will be useful while integration of process and variables with the simulator daemon processes. After integrating and collecting the data from simulator, how it is updated into the screens is also discussed. At the end, development results are depicted.

Chapter 5 covers integration of the process with the simulator. It discusses registering and de-registering of the process. Saving the state of the simulator is discussed. An example of fully developed and integrated panel 1 is shown.

Chapter 6 covers the methodology of reusing the framework developed to create virtual panels.

Chapter 7 covers the summary of the project and future work that can be done with the project.

Chapter 2

Hardware and Software Architecture of the Simulator

Hardware and Software form a crucial part of computer based system. Understanding of system is incomplete until the hardware-software interlink is not analysed. This chapter provides an insight on the hardware and software architecture of the PFBR simulator. It includes the components and interactions among them.

2.1 Hardware Architecture of PFBR Simulator

Hardware of PFBR simulator provides an environment of the control room to the operators for training and qualification. The hardware of the simulator can be broadly classified into following subsystems. Each of the subsystems is discussed in subsequent sections.

1. Simulation Computer
2. Control Room - Panels and Consoles
3. I/O System
4. Local Control Centres
5. Data Highways

6. Instructor Station
7. Process Computers

2.1.1 Simulation Computer

Simulation computer is heart of the total simulator system. It executes numerous codes representing the mathematical models of the various subsystems of plant. It updates the database in the plant computer for providing the information in the display stations.

2.1.2 Control Room - Panels and Consoles

Simulator panels and consoles are replica of the main plant control room. The layout and position of devices on panels and consoles is identical. Operator controls the reactor from the console. Panels are having all the information for the system. Console contains minimal information that is required to control the reactor and take immediate actions. Console and panel contain many components which are redundant. For the sake of simplicity and avoiding confusion term “panel” is used in context of virtual panels for both the consoles panels.

2.1.3 I/O System

The I/O system corresponding to each simulator panel is housed in the electronic cabinet behind the panel itself. The number of Analog Input, Analog Output, Digital Input and Digital Output interface cards in each I/O system depends on the number of signals handled in the respective panels. I/O system consists of three building blocks, I/O controller, I/O bin and power supply units.

The I/O controller is an industrial grade computer which consists of suitable interface boards for communicating with various analog and digital I/O boards. I/O bin consists of four varieties of I/O boards, viz., Analog Input, Analog output, Digital Input and Digital output boards. These boards interface with the control panel devices data from analog to digital and

digital to analog wherever required and exchange data with the I/O controller. Each of the I/O systems has an instrumentation power. Figure 2.1 depicts the I/O system and its interactions with the other hardware.

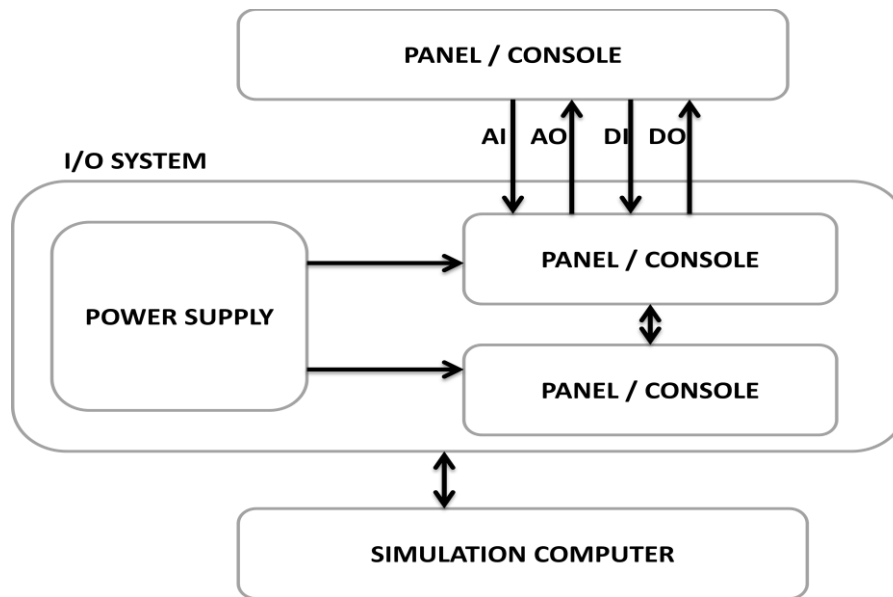


Figure 2.1: I/O System and its Interactions

2.1.4 Local Control Centre (LCC)

Local control centres (LCC) are simulated by computers. LCC gets signal from field in the actual plant but in simulator, they get simulated field data depending on the reactor state from the simulation computer.

2.1.5 Data Highway

All systems located in the LCCs transmit data and messages to control room through a dedicated data highway. Data highways are dual Local Area Networks (LANs). Ethernet

switches in LCCs receive data, once every scan cycle from the systems and forward to the data highway. Display station display data and messages received.

2.1.6 Instructor Station

The instructor station is used by instructor to monitor the various operations being performed by the trainee, create various incidents/malfunctions for training. During the training session, the simulator operations are monitored and controlled through the instructor station.

2.1.7 Process Computers

The process computers receive plant information from simulation computer, performs processing, updates plant parameters database, and store the data. Graphic display stations on operator console and panels receive data from process computers and displays in formats like mimic diagrams, bar graphs, soft graphs, and soft alarms.

2.2 Software Architecture of PFBR Simulator

The simulation computer executes software codes representing mathematical models of various sub-systems of PFBR in real-time. The simulation software components are explained in following sections. Plant computers (PC) receive plant data from the simulation computer, process them and update in database. This plant information is fed to the display stations on the console and panels for viewing in different formats. The panels and operator consoles are replica of the PFBR control room with the same devices and layout. The hardwired systems are interfaced with simulator using I/O systems handling analog and digital signals. Local control centre (LCC) refers to the instrumentation and control systems simulated using actual hardware or alternative computer systems.

Simulator software system organisation

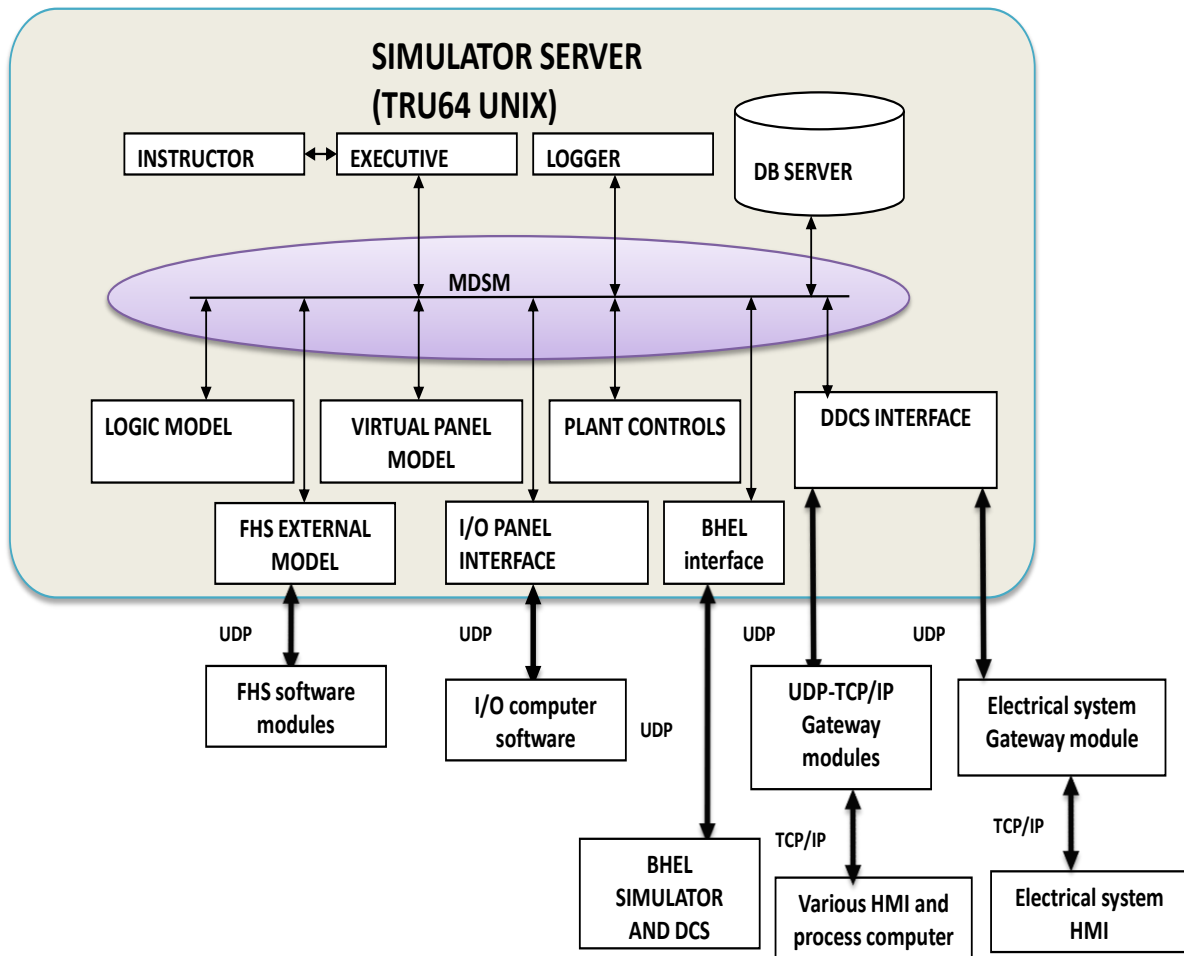


Figure 2.2: Organisation of simulator software components

Organisation and interactions between the simulator software components is depicted in figure 2.2. MDSM forms a virtual intermediate layer using which all the simulator data is communicated among the processes. Components of simulator explained in this chapter are core components. Some of the components are added externally and are covered in chapter 3.

All the processes registered with the MDSM use the MDSM functions or the IPC wrapper implemented on top of MDSM functions. IPC wrapper abstracts the direct use of MDSM functions and hides the complexity. Two components can interact with each other using only through MDSM. Database access is also done using the MDSM functions. Figure 2.2 depicts the communication between the components of the simulator.

Simulation software provides simulation model libraries, tools to build and integrate components and run-time environment for running simulator in real-time. It includes the following application specific tools in addition to core component of simulator – Instructor, Executive, Database server, IC logger, Messaging and Data Sharing Mechanism.

- Process Modeller – For simulating the plant process models
- Logic Modeller - For simulating process controls
- Virtual Panels - For simulating the hardware panels

Following sections explain in detail the software components of the simulation software.

2.2.1 Daemon Processes

A daemon [31] is a long-running background process that answers requests for services. Daemon processes are often started when the system is bootstrapped and terminate only when the system is shut down. Because they don't have a controlling terminal, they run in the background. They are started by the user in PFBR simulator.

There are three daemon processes in simulator architecture namely Database Server, Logger and MDSM each of which is explained in subsequent sections.

2.2.1.1 Database Server

It provides a single database with uniform functionality for all the components of the simulator. It supports multi-user access to the database and the processes are independent of the database implementation. In the development environment multiple projects can be supported by a single database.

The user interface for the executive process is provided by *instructor* process. The instructor provides facilities to control and monitor the operation of the simulator and conduct training sessions. It displays menu, status information, alphanumeric data and graphical information related to simulator, operator actions and training. It provides access to the complete simulator database. It can display variety of information either transiently upon user demand or continuously as required.

2.2.1.2 MDSM

MDSM stands for *Messaging and Data Sharing Mechanism*. MDSM consists of two loosely coupled components - messaging mechanism and data sharing mechanism.

Messaging allows UNIX processes to talk to each other by sending and receiving binary messages. It allows a group of processes tuned similarly to talk to each other both within machine boundaries and across them. Data sharing part allows the applications to publish binary data and read them. Each data item has a name and is recognized by this name only.

The main functions of MDSM are:

- Managing blocks of shared memory.

- Initializing shared memory location to a value supplied by the subscribing process
- Maintaining tables of published and subscribed variables along with their locations.
- Managing message passing between all the processes that are registered with it.

It is essential for the simulator operation that various process modules of the simulator communicate with each other by exchanging messages and data.

Various API calls made to the MDSM daemon are described in the subsequent sections.

2.2.1.2.1 Callback Function

The application must provide a callback function [25] which will be called every time an event occurs in the system. The event can be either arrival of a message or an error or an event within the MDSM (i.e. connection dropout).

The prototype of callback function is as follows:

```
void callback (int msgType, MDSM_ProcId *id, void *msg, int len);
```

where:

- *MDSM_ProcId* – an internal MDSM structure used to uniquely identify the process
- *msgType* – indicates the message type

- *msg* – a pointer to a binary buffer containing the message received
- *len* – the size of the buffer

MDSM invokes the callback function on a number of occasions, including errors, receipt of a message, new process attaching to the system or detaching from it. The reason for calling is indicated by the *msgType* parameter. The values it can take are listed in the table below:

Table 2.1 MDSM Message Types

<i>MDSM_MSG_RX_TAG</i>	A message has been received from another process
<i>MDSM_CONN_OPENED_TAG</i>	A new process has attached to the system
<i>MDSM_CONN_CLOSED_TAG</i>	A process has disconnected from the system
<i>MDSM_ERROR_TAG</i>	An error has occurred within MDSM or has been detected by it

2.2.1.2.2 Registering a Process

Before working with a process you need to register it with MDSM. To do so, following call is used:

```
MDSM_ProcId *MDSM_RegisterProc(
    char *processName,
    void(*callback)(MDSM_ProcId *id, char *msg, int len);
);
```

where:

- *processName* – the logical name of the process i.e. “VPanels”
- *callback* – pointer to the callback function
- return value – the process identifier constructed by MDSM

2.2.1.2.3 Registering and De- Registering the Variables

A data item (a variable, a signal) needs to be registered in order for it to become available for other processes. De-registration of a variable causes MDSM to ‘forget’ about it, at least forget about the association of the variable to the current process. When a variable is not de-registered, MDSM keeps updating that variable in the shared memory of the process. It leads to waste of resource and effort performed by the MDSM.

2.2.1.2.3.1 Registering for Publishing

If a data item is going to be published, it is registered using *MDSM_RegisterVarByValue* function:

```
int MDSM_RegisterVarByValue(char *varName, void *addr, int len, int switch, void *defValue );
```

where:

- *varName* – a text string containing the name of the variable, under which it will be known

- *addr* – the address in the application memory containing the variable
- *len* – the size of the variable
- *switch* – indicates whether the variable will be published or subscribed to. For publishing it must be set to *MDSM_PUBLISH*
- *defValue* – It is not used when publishing and should be NULL
- return value – indicator of success, 0 indicates success

An attempt to publish a variable that is already published by another process causes an error.

2.2.1.2.3.2 Registering for Subscription

The variables that are subscribed can be passed by value or by reference. Those variables that are passed by value will be copied from the shared memory to the local memory by MDSM. The referenced variables will reside in the shared memory and will be read by the application code, when required.

To register a variable for subscription by reference use the *MDSM_RegisterVarByReference* function:

```
int MDSM_RegisterVarByReference(char *varName, void **addr, int len);
```

To register a variable for the subscription by value, use the *MDSM_RegisterVarByValue* function with the *MDSM_SUBSCRIBE* switch.

```
int MDSM_RegisterVarByValue("myfloat", &myfloat, sizeof(float), MDSM_SUBSCRIBE,  
  
defFloat(25));
```

where:

- *varName* - Name of the variable with which it is registered
- *addr* - Address of the variable
- *len* - size of the variable

The last parameter *defFloat* is used to specify the default value. There is a family of *def* functions which allows to specify defaults for various types:

- *defFloat* – floating point values
- *defInt* – 32-bit integers
- *defLong* – 64-bit integers
- *defStrn* – C-strings

These are defined in the MDSM code. For specification of defaults for the other types, the existing functions can be used as template for the new ones.

The default values are used when the variable is not published. The system writes them to shared memory while registering the variable and copies them into local memory every time *MDSM_ReadAll* gets called.

Functions *MDSM_RegisterVarByValue* and *MDSM_RegisterVarByValue* return a success/failure code 0 in for success and a non-zero error code for failure.

No error is generated if a variable is subscribed to but not published. When a variable is subscribed to, MDSM allocates the shared memory for it and writes the default value into that area. If no default value is specified, the result is unpredictable. To indicate that the variable is not published, the registration functions set a global variable, *MDSM_WarnCode* to 1. This variable can be used by the simulator code to check the publishing status immediately after a call to *MDSM_RegisterVarByValue* or *MDSM_RegisterVarByReference*. Multiple subscriptions for the same variable are allowed, including multiple subscriptions by the same process.

2.2.1.2.3.3 De-Registering the Variables

Variable is de-registered after its usage is over. It can be done in two ways:

- Function *MDSM_DeRegisterVar(char *varName)* can be used to de-register individual variables.
- Function *MDSM_DeRegisterAll()* de-registers all the variables registered before.

MDSM never really 'forgets' completely about variables that were once registered. While de-registering the variable, the association between the variables and the process is discarded. However, the memory remains allocated and the system remembers the variable length. This information is only destroyed when the last process in the simulator exits and the shared memory is returned to UNIX. If a process attempts to re-register the same variable as before with a different length, it will cause an error.

2.2.1.2.4 Publishing Data

Publishing data is done in response to the 'Publish' message from the executive. Publishing during the computation cycle is inherently dangerous and may result in unpredictable system behaviour.

The following functions may be used to publish data:

```
int MDSM_PublishAll();
```

This publishes all data registered by the current process for publishing. It will return 0 in case of success and non-zero value in case of failure.

Alternatively, *MDSM_PublishValue* function can be used:

```
int MDSM_PublishValue(char *varName, void **addr, int len);
```

This publishes a specified variable. Extreme caution is taken before using *MDSM_PublishValue* function. It can be invoked after all other processes have already published their data via *MDSM_PublishAll()*, but before the computation has commenced. Incorrect use of it may result in inconsistency in data. *MDSM_PublishValue* is slow as compared to *MDSM_PublishAll*.

When *MDSM_PublishValue* is invoked for a variable, function allocates space in shared memory if it is not done before. However, it does not mark the variable being published by the current process, so that the calls to *MDSM_GetItemInfo()* or *MDSM_GetFirstSymb()*, *MDSM_GetNextSymb()* will not find any association between the variable and the process.

2.2.1.2.5 Fetching Data

Fetching data is done at any time during computation cycle, but not during publishing.

One of the two functions is used for fetching data:

```
int MDSM_ReadAll();
```

It reads all variables that have been subscribed to.

```
int MDSM_ReadValue(char *varName, void *addr, int len);
```

It reads only one variable into the specified address, *MDSM_ReadValue* returns the MDSM error code as shown in table (write number) or 0 in case of success or -1 if the variables is not published. When *MDSM_ReadValue* is invoked for a variable, the function allocates space memory if it has not been done before. If a variable is not published by any process, the *MDSM_WarnCode* variable will be set immediately to 1 after the call to *MDSM_ReadValue*.

2.2.1.2.6 Sending a Message

To send a message to all processes, function *MDSM_Broadcast* is used.

```
int MDSM_Broadcast(void *msg, int msgLen, u_int8 priority);
```

This broadcasts the message to all the processes those are visible from the current one.

- *msg* - pointer to a binary buffer containing the message

- *msgLen* - length of that buffer
- *priority* - the message priority, (0,255). Higher the number, so the priority

To send a message to a specified process, use function:

```
int MDSM_Send(MDSM_ProcId *id, void *msg, int msgLen, u_int8 priority);
```

The first parameter *id* must point to *MDSM_ProcId* structure identifying destination process. *msg* and *msgLen* are message and length of the message respectively.

2.2.1.2.7 Receiving a Message

MDSM doesn't provide asynchronous inter process communication. To enable message processing, process has to poll for input using *MDSM_PollInput*;

```
int MDSM_PollInput(int how);
```

Parameters can be:

- *MDSM_FOREVER* - don't return until a message has been received
- *MDSM_NOWAIT* - check if there is any message and return immediately if no message.
- *a positive value* - the maximum delay, in milliseconds, to wait for in the select call.

This function checks if there are any pending messages and process them. The messaging mechanism is based on the select system call. If there is a file descriptor ready which is not associated with *MDSM*, the function will return it. Otherwise, a zero will be returned.

For each message, *MDSM_PollInput* will invoke the registered callback function, passing the process id and message to it.

2.2.1.2.8 De-Registering the Process

Before a process terminates it must de-initialise using *MDSM_DeRegisterProc*

int MDSM_DeRegisterProc();

will de-register the process the *MDSM*.

It closes all the connections and removes entries in the *MDSM* variables table that are related to the process.

To deregister a single variable *MDSM_DeRegisterVar*

*int MDSM_DeRegisterVar(char *varName);*

where *varName* is the name of variable with which it was registered with *MDSM*.

2.2.1.3 Logger

Logger provides a uniform, centralized mechanism to save and restore information about the state of the simulator and user input. Simulator processes need to save and restore states upon demand at periodic intervals. The simulator processes will employ the logger via Logger API to save and load their information and will have no direct access to underlying files and other structures. The logger executes operation on behalf of the process and reports

back the completion status. The simulator processes act according to the commands from the executive.

The information saved or restored by the logger can belong to one of the four types: Initial Condition (IC), Snapshot, Backtrack and Replay. Parameters from the logic processes are managed by the logger. The information is organised into bins, with each bin having ID, which together with the bin type, uniquely identifies a bin. The logger provides the following basic functions to processes:

- Saving data into a bin.
- Reading data from a bin.
- Listing available data bins of a given type and retrieving their heading information.
- Deleting a data bin.
- Converting data bin of one type to another.

2.2.2 Process Models

The process models in simulator replicate the plant using mathematical techniques. All conventional sub-systems of the plant - hydraulic, steam, air/gas and electrical as well as non-process elements such as actuators and transducers are modelled using promodeller. This tool is used to simulate the plant process models organised as an integrated set of functional units:

- Graphical User Interface (GUI)
- Plant and Simulation Data Structures
- Database Manager
- Simulation Model Libraries
- Simulation Configuration Builder

2.2.3 Plant Controls

Plant controls are software components which implement specifically designed control logics and emulate various control systems in the plant. Plant controls are simulated using a menu driven logic design graphic tool. The tool supports emulation of plant control systems, including:

- Modulating loops
- Mode changing and status supervisory logic
- Interlocking logic
- Sequence controls
- Alarm formations

It provides programmer with the facility to realise any required control logic by drawing logic sheets using the standard gates library and also to create and use application specific function blocks. It has a graphical drawing editor for designing the schematics of process control and automatic code generator for generating code (in C language) for the logic sheets designed. The tool checks for error both during drawing phase as well as during the automatic code generation. The output of the code generator will compile and link directly into the simulator environment.

Chapter 3

Interaction between Models of the Simulator

This chapter discusses about external models and their inclusion mechanism in the simulator system. External models follow steps to be a part of the system. Steps for initialisation and run time phase are discussed in this chapter. Interaction between the simulator components is discussed with pictorial representations.

3.1 External Models

External models are foreign process models which are developed from scratch to work in the simulator environment. For example, nuclear sub-systems of PFBR *cannot* be modelled using the process modeller as it supports only conventional components. Hence mathematical models of the non-conventional nuclear systems need to be developed and integrated with the other software components of the simulator. External models developed comply with the specifications of simulation software component in order to run in the simulator environment. The external models can be developed using C or FORTRAN language.

A foreign module code is available to include the initialisation and run-time blocks and make use of the Inter Process Communication (IPC) [31,32,33] handler Application Programming Interface (API) to fulfil the requirement of simulator components. It provides functions for registering the variables for publishing, subscribing, running a process under simulator environment, etc. The IPC handler hides the complexity of communication and synchronisation with centralisation for debugging in the distributed architecture. At its helm, it uses the MDSM functions as described in the section 2.2.1.2.

Some of the examples of the external models developed are Neutronics System, Reactor Core, Primary Sodium System, Secondary Sodium System, Secondary Grade Decay Heat Removal System and other PFBR specific systems.

Virtual panels are developed as an external model. The virtual panels in the alpha based simulator are developed using proprietary tool which is available only for Tru64 Unix. It is not portable to Intel based simulator. Virtual panels for the Intel based simulator are developed using Qt and SQLite.

The proprietary graphics software package provides facility to model screen based panels with range of devices, similar in appearance and function to the range of industrial controls and instrumentation. Package includes graphical tool editors, runtime function libraries, the converters and other tools. The graphical editor allows the developer to create models with primitive graphics objects like lines, rectangle, circles, curves and pre-defined sub-models like lamps, button, meters, symbols that can change in real-time in response to the changes in the application data.

3.2 Basic Operation of Simulator Component

Each process or component of simulator system performs the following steps when the simulator runs.

3.2.1 Initialisation

Initialisation of a simulator component involves following steps. Figure 3.1 describes the flow diagram of the initialisation of a component.

- Register itself by name with MDSM and the Executive
- Wait for Executive command : register variables to be published
- Register all the variables to be published
- Wait for Executive command : register variables to be subscribed
- Register all the variables to be subscribed

Figure 3.1 shows the flow diagram for the initialisation of simulator component.

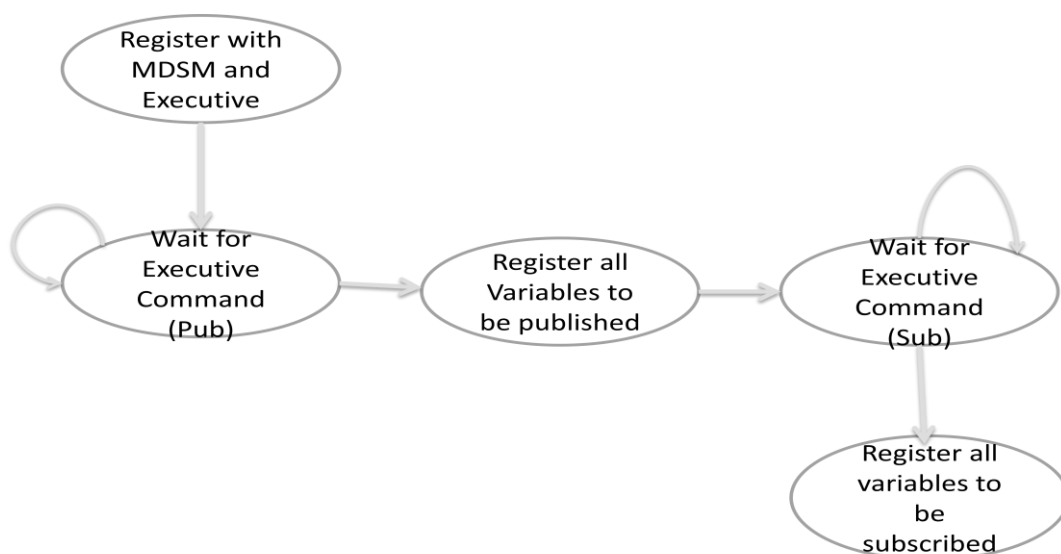


Figure 3.1: Flow Diagram for Initialisation of simulator components

3.2.2 Run Time

Run time behaviour of the component should follow the sequence as described in following steps, and Figure 3.2 depicts the flow diagram for run time behaviour of the component.

- Wait for the start of the cycle
- Read all the subscribed variables
- Perform calculations and update the outputs
- Advise Executive when the calculations are complete
- Wait for the publish command from the Executive
- Publish all the outputs

Figure 3.2 shows the flow diagram for run time behaviour of simulator software component.

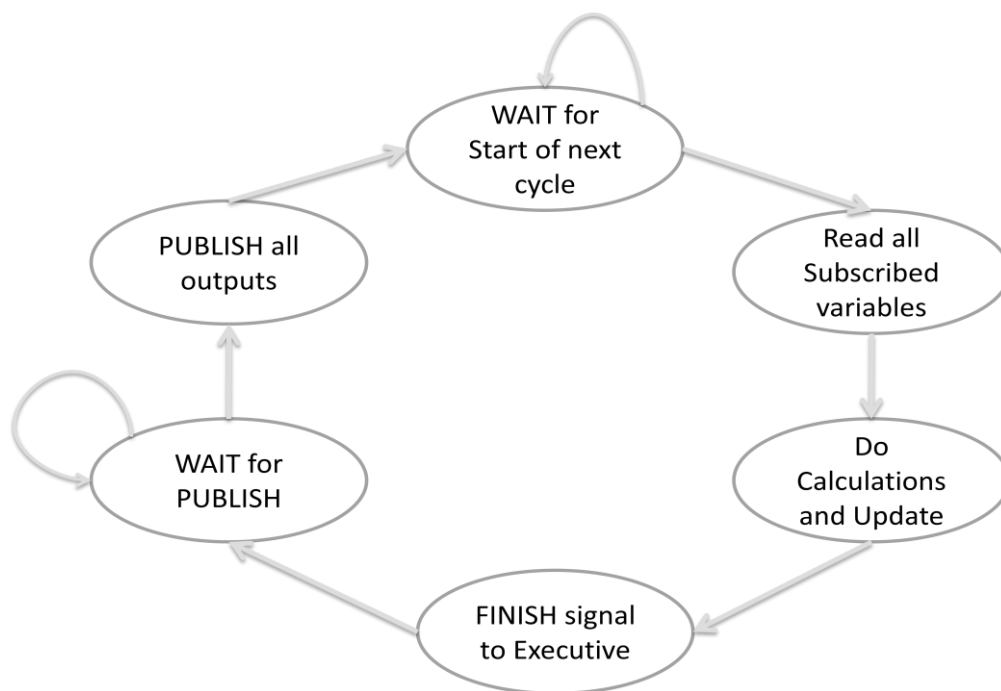


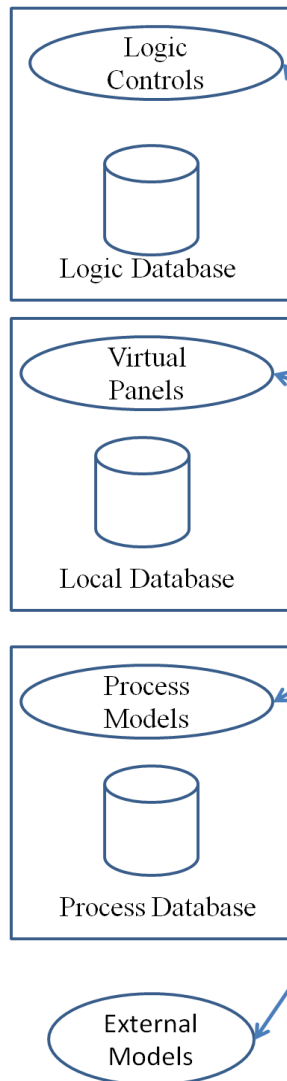
Figure 3.2: Flow Diagram for Component Runtime Operation

3.3 Interaction between Simulator Components

The interaction between the components of simulator environment is achieved with the help of MDSM and a global simulator database maintained by database server of the simulator. The local data generated by a simulator component is integrated into the global database to allow other simulator components to access the data. MDSM uses shared memory to store the messages and data exchanged between components. The interaction among the components is depicted in the Figure 3.3:

MDSM coordinates between the processes through inter-process communication. It is accomplished by exchange of messages. Every component implements a certain set of callback functions in order to execute their tasks. Logic control has its own database in which the logic diagrams are stored. If any of the two components require exchanging messages or data, they need to communicate through MDSM. Mode of communication is inter-process communication using the messages. For virtual panels, shared memory inter process communication is also used when the variables are subscribed and published. Process models store their data in the local database, where all the computations as well as required information is stored. Executive communicates with the MDSM and other processes to instruct the actions to be taken in each cycle. Instructor provides a user interface to the executive process and handles the front end part of executive. It also interacts with the operator instructor for training purpose. Insertion of malfunctions and different plant scenarios can be simulated from the instructor process.

Models Developed



Core Components

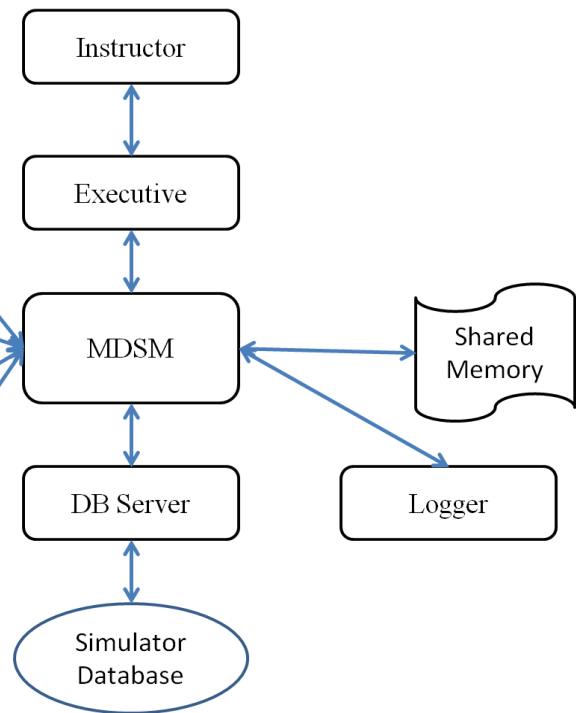


Figure 3.3: Simulator Software System and its Components

Chapter 4

Development of Framework to Create Virtual Panels

This chapter discusses details related to development of framework to create virtual panels. It discusses the methodology used for development. Also, it discusses the information and data stored in the database that will be useful while integrating the process and the variables with the daemon processes. After integrating and storing the data from simulator, the mechanism implemented to update the virtual panels is discussed.

4.1 Virtual Panels

Virtual panels are soft panels which emulate control panels with animated panel equipment widgets to virtualize operator interface. Widgets show real time behaviour for changes in application data. These include alarms, pushbuttons (momentary and latching type), selector switches, gang-selector switches, LEDs, display segments like LCD, bar graph indicators, ammeters, voltmeters, synchroscope, etc.

Virtual panels provide a benefit that all panels and consoles can be accessed from a single place, rather than physically moving between the panels and consoles like that for

hardware panels. During the development phase of simulator, when each test is not possible on hardware panels, virtual panels are used for testing the data from the other models. Access to data is done in similar way as for the hardware panels.

4.2 Tools used for Development

Tools chosen for development are open source. Open source software provides inherent advantages. It is available for free with a large community support. Open source software is easily available which reduces the development time. Tool chosen for virtual panel development is Qt with database support of SQLite. Further sections describe the feature of the tools used and their advantages.

4.2.1 Qt

Qt is a cross-platform application development framework for desktop, embedded and mobile. Supported Platforms include Linux, OS X, Windows, VxWorks, QNX, Android, iOS, BlackBerry, Sailfish OS and others.

Qt is not a programming language on its own. It is a framework written in C++ [41,42]. A pre-processor, the MOC (Meta-Object Compiler)[45], is used to extend the C++ language with features like signals and slots. A signal is a function that is invoked an action takes place. It is used to detect the action. A slot is a function that defines tasks to be performed after the action has taken place. For example, click on a button is an action and displaying some content on click is a task. Therefore, click is a signal and display is a slot. Before the compilation step, the MOC parses the source files written in Qt-extended C++ and generates standard compliant C++ sources from them. Thus the framework itself and

applications/libraries using it can be compiled by any standard compliant C++ compiler like Clang, GCC, ICC, MinGW and MSVC. Qt is available as free software under several versions of the GPL [7] and the LGPL [8].

Qt provides visual elements commonly known as widgets. Widgets are simple objects which can be framed as a real world entity. Real world entities can be created using a combination of the widgets. For example, pushbutton is a simple widget. It can be a momentary pushbutton or latching type pushbutton. It can be a display entity when its click option is disabled. Combination of pushbutton can be used to implement selector switches which are quite common in power plants. Implementation of the components will be described with the components in the subsequent sections.

Each widget has properties associated with it. All the information about the widget is stored into the properties. For example, name is a widget property. Text to display is a property. Dynamic properties can be given to the widgets as per the requirement. These properties can be modified from the GUI as well as from code. For implementation of several components dynamic properties has played an important role for identification of the components.

4.3.2 SQLite

SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. Self-contained database is the one which requires support and resources only from the host computer. Such databases do not require setting up of server for them. The entire database is integrated into whatever application needs to access the database. The only shared resource among the applications is the single database file as it sits on disk. By eliminating the server, a significant amount of complexity

is removed. Thus it requires no configuration with ready to use feature. Transactional databases are those which support handling of multiple statements to accomplish a task or a transaction. The code for SQLite is in the public domain and is thus free for use for any purpose, commercial or private. SQLite is the most widely deployed database in the world with more applications than we can count, including several high-profile projects.

SQLite is an embedded SQL database engine. Unlike most other SQL databases, SQLite does not have a separate server process. SQLite reads and writes directly to ordinary disk files thus reducing the access time and working as an in memory file for an application. A complete SQL database with multiple tables, indices, triggers, and views, is contained in a single disk file.

4.3 Basis for selection of tools of development

The tools chosen for development are Qt and SQLite. Both are chosen mainly because they are open source. Apart from being open source, there are many other advantages that they provide. Comparison between Qt and a similar open source real time software GLG toolkit is listed in table 4.1.

SQLite is one of the most popular in-memory databases. Most of the other in-memory databases are either proprietary or setting them is more effort seeking than SQLite. SQLite is easiest of them to setup. It consists of a file, which handles all the relational queries. Due to the in-memory feature, it is faster than the traditional relational databases.

Table 4.1 Comparison between Qt and GLG Toolkit

S.No.	Qt	GLG Toolkit
1.	Open Source	Open Source (only community version)
2.	Cross Platform	Cross Platform
3.	Useful for graphics and logic building	Useful for graphics and logic building
4.	Supports C, C++,JAVA, C#	Supports C, C++,JAVA, C#
5.	Supports SQL database access, XML parsing and thread management	No such support

Qt has an advantage over GLG [46] that it provides database access, XML parsing and management of threads. This can be used in future for improving the architecture of the virtual panels by proper distribution of tasks among the threads. One more advantage that Qt is having is that GLG is open source only for the community version and not for commercial purposes.

4.4 Types of Signals

There are four kinds of signals commonly used in industrial plant environment - Digital Inputs (DI), Digital Outputs (DO), Analog Inputs (AI), and Analog Outputs (AO). Digital Inputs (DI) are the digital signals received by simulation computer from the panels or any other node in the network. Digital Outputs (DO) are digital signals sent by simulation computer to panels or other nodes in the simulator. Analog Inputs (AI) are the analog signals received by simulation computer from the panels or any other node in the network. Analog

Outputs (AO) are analog signals sent by simulation computer to panels or other nodes in the simulator. Panels mostly deal with DI, DO and AO. The components in the simulator emit/receive one of the signals as mentioned below.

Digital Input (DI) - Pushbuttons (Momentary/Latching), Selector Switches (Normal, Key-Selector and Gang Selector),

Digital Output (DO) - Alarms, Light Emitting Diode (LED), Semaphore indicators

Analog Output (AO) - Seven segment displays, Bar graph indicators, analog meters, synchroscope.

4.5 Development Methodology

A console or panel consists of various components. Components include alarm annunciations, pushbuttons, selector switches, gang selector switches, and various meters for display. These components are developed to completely provide the functionality of a hardware panel. Changes made through virtual panels are same as that of hardware panels. In addition, virtual panels should keep track of the changes in the hardware panels. When loaded for the first time, virtual panels capture the state of the hardware panels and display same status as that of hardware panel. Also while exiting, the state of the virtual panels is saved so that the next time they are loaded again then status of the components should be displayed same unless, some change occurs in the hardware panels. This saving of state and loading again is provided by the simulator through instructor. The state of the virtual panels is saved through the data structures used for storing the variable values of variables. The data structures used in the virtual panels development are explained in next section. All the components developed are incorporated in a template panel from which, components can be

taken and copied or inserted into other virtual panel under creation. The variables of the component inserted in the panel needs to be modified to realize the behaviour according to the corresponding hardware panel component.

4.6 Details of Data Structures Used

Data structures [47] are the most important entity of any software. Data structures provide organisation and storage format that enables efficient access and modification. More precisely, data structure is a collection of data values, the relationships among them and the functions that can be applied to the data. This section describes all the data structures used during different phases of virtual panels.

During the first time loading of virtual panels, data is stored in database DB3 and SQLite. DB3 is used to communicate data to simulator and SQLite to convey information about the GUI components. SQLite database contains a single table *widgetdata* with description as shown in Table 4.2.

DB3 is an in-memory database. It has six fields which are similar to the above database.

varname - It is name of variable with which the widget is associated.

varflagname - It is flag variable name for the widget.

varstatus - It describes the type of variable (DI,DO,AI,AO).

varnamesys - It is similar to sysname of SQLite database.

offset - It is offset of the variable in database.

offset size - It is size of the offset.

Table 4.2 Table for Storing Widget Information

Column Name	Type	Description
Tagname	varchar(24)	It is the primary key that identifies each and every widget uniquely.
Sysname	varchar(24)	It is the system name of the variable associated with the widget.
Flagname	varchar(24)	It is a variable that is used to identify different usages for the same type of widget.
Status	varchar(24)	It describes the type of variable (DI,DO,AI,AO).
Offset	varchar(10)	It is offset of the variable in database.
Offsize	varchar(5)	It is size of offset
Comment	Integer	It is the index of variable in the database through which access to value of variable is done.

Dynamic properties with similar names as that of database fields is assigned to each widget, accessed and used when required. Data is then segregated in form of different components of the panels and stored as the virtual panel data. For saving data following data structure is used.

It stores all the information pertaining to the screens including the number of binary and analog components in the screens. MAX_VAR is number of maximum number of variables in the screen. It is a user defined parameter. All the variables with their name are stored in *PD_INFO* structure for further interacting with the simulator.

```

typedef struct
{
    int numAnalogVars;
    int numBinVars;
    int numFlagVars;
    int numAnlFlagVars;

    char analogVarName[MAX_VAR][32];
    int analogVarInd[MAX_VAR];
    int analogVarSendFlag[MAX_VAR];
    int analogVarUpdateFlag[MAX_VAR];

    char binVarName[MAX_VAR][32];
    int binVarInd[MAX_VAR];
    int binVarSendFlag[MAX_VAR];
    int binVarUpdateFlag[MAX_VAR];

    char flagVarName[MAX_VAR][32];
    int flagVarInd[MAX_VAR];
    int flagVarSendFlag[MAX_VAR];
    int flagVarUpdateFlag[MAX_VAR];

    char anlFlagVarName[MAX_VAR][32];
    int anlFlagVarInd[MAX_VAR];
    int anlFlagVarSendFlag[MAX_VAR];
    int anlFlagVarUpdateFlag[MAX_VAR];

    int initFlag;
}PD_INFO;

```

Messaging and Data Sharing Mechanism (MDSM) requires a local memory address for the variables of the process when subscribed by value. A data structure

MOD_VARS_INFO is created in which simulator writes/read values of subscribed/published variables. The address of the variables of the structure is passed while registering of variables.

A variable can have a binary value or an analog value, thus a union is created in which only either of the type can be stored. *WORD* is defined as integer type. *offset* refers to the index of variable with name *varNameSys*. *offs_size* is the size of the *offset* (*WORD/float*) *varStatus* defines the type of variables, either it is for subscription or publishing.

```
typedef struct {
    union VAL
    {
        WORD binValue;
        float analogValue;
    }Value;
    unsigned int offset;
    unsigned int offs_size;
    char    varNameSys[25];
    char    varNameGms[25];
    char    varFlagNameGms[25];
    char    varStatus[4];
} MOD_VARS_INFO;
```

Apart from storing the virtual panel components data, components are categorized into a number of different structures according to their behaviour. Three types of categories are used namely, Pushbutton, Selector Switch and Toggle Switch. Structure for pushbutton is described below.

```

typedef struct
{
    int type;
    char pbVarName[24];
    int pbVarInd;
    WORD pbVarVal;

} PBS;

```

The *type* stores type of component. The information for the *type* is mentioned in Annexure 1. *pbVarName* stores the name of the component with which it will be registered with *MDSM*. *pbVarInd* is the index of pushbutton in *modvarsinfo* structure and *pbVarVal* stores the value of the pushbutton.

Structure of selector switch is described below.

```

typedef struct
{
    int type;
    int num_pb;
    char ssVarName[MAX_POS][24];
    int ssVarInd[MAX_POS];
    WORD ssVarVal[MAX_POS];

} SSS;

```

The *type* stores type of component. The information for the *type* is mentioned in Annexure 1. *num_pb* stores number of positions of the selector switch. *MAX_POS* is the number of maximum positions that a selector switch can have. *ssVarName* stores the name of the component variables with which it will be registered with *MDSM*. *ssVarInd* is the array of indexes of switch selections in *modvarsinfo* structure and *ssVarVal* stores the value of all the positions of the selector switch.

Structure of toggle switch is described below.

```
typedef struct  
{  
    int type;  
  
    char lVarName[24];  
    int lVarInd;  
    WORD lVarVal;  
  
    char mVarName[24];  
    int mVarInd;  
    WORD mVarVal;  
  
    char rVarName[24];  
    int rVarInd;  
    WORD rVarVal;  
  
} TGS;
```

Toggle switch has three positions (left, middle, right) and it toggles its positions immediately it is released. The *type* stores type of component, the information for the type is mentioned in Annexure 1. *lVarName*, *mVarName* and *rVarName* stores name of variables for three positions. *lVarInd*, *mVarInd* and *rVarInd* store the index of the corresponding position. *lVarVal*, *mVarVal* and *rVarVal* store the value of the positions.

4.7 Development of Panels

Development of panels begins with the top level welcome view consisting of pushbuttons to open all the virtual panels. Top view provides access links to virtual panels. It is depicted in the Figure 4.1.

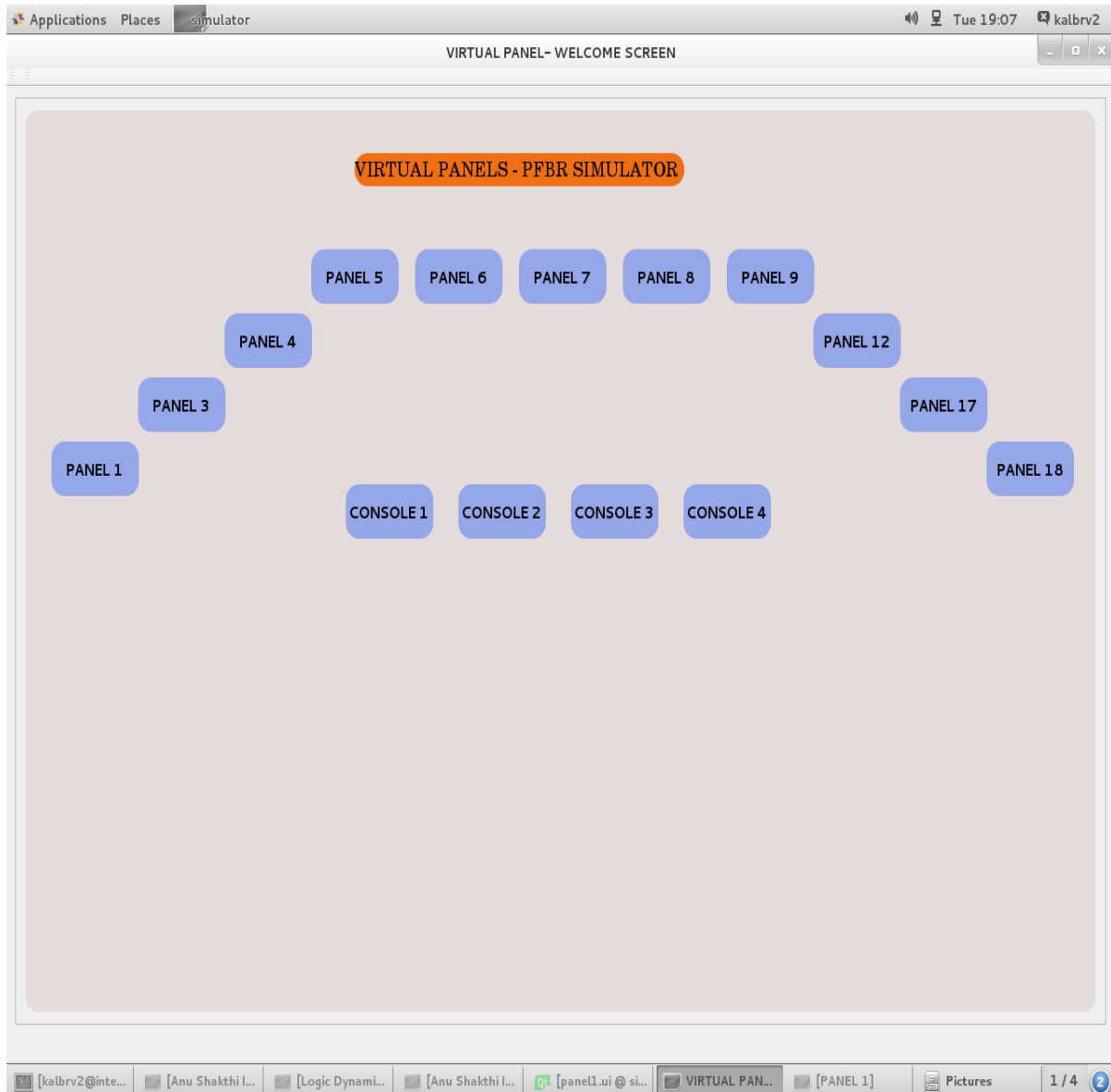


Figure 4.1: Top level view of Virtual Panels

Each button corresponds to a virtual panel and clicking the button opens the respective panel. For example, clicking button "PANEL 1" prompts to open virtual panel

created for panel 1 as depicted in Figure 4.2. Panel 1 consists of three tabs namely, Alarms, Displays and Controls. Figure 4.2 shows the Alarm tab opened. Alarms tab has all the alarms in the panel 1. Control and Display tabs encapsulate the display items such as LED and control items such as selector switches, pushbuttons etc. After the “PANEL 1” is opened, tabs can be switched by clicking on the name of the appropriate tab. More than one virtual panel can be opened for the single panel showing the synchronous behaviour due to the centralised nature of database.

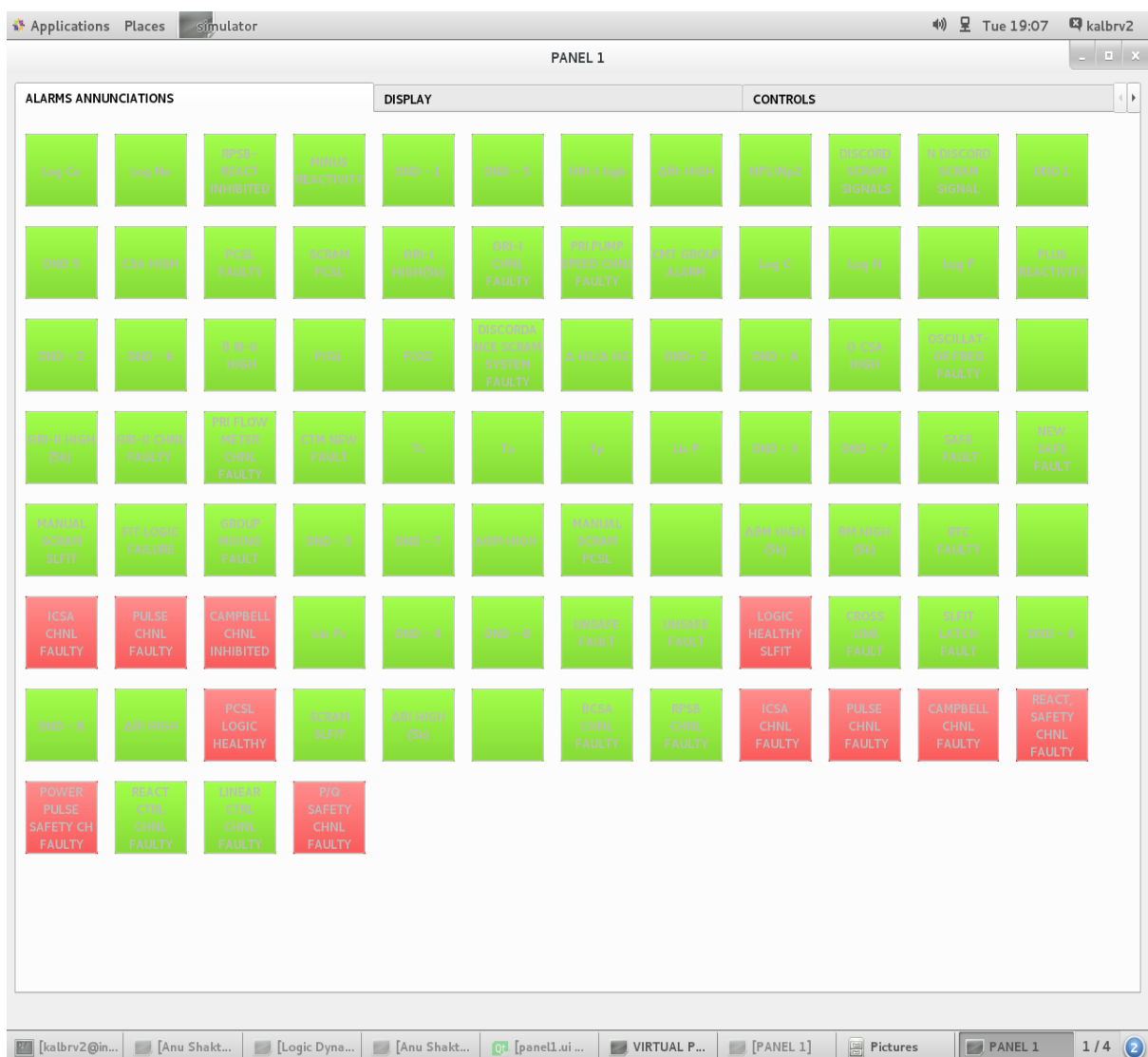


Figure 4.2: Alarm Tab of Panel 1

4.8 Panel Components and their Implementation

The major components of panels are alarms, display components such as LED, 7-segment etc., pushbuttons (momentary and latch type), selector switches (key selector, normal selector, gang selector). Each type of component has its own properties and actions. Components and their types with their pre-processing data are described in the further sections. One example for each type is described for the clarity. Results for each type are shown in the next sections and final consolidated results are shown in section 4.13.

4.8.1 Alarm

The top portion of hardware panels is dedicated to alarms. Alarms are visual annunciations that indicate any deviation from the normal operation of the system. Alarms are the first indications that notify the operator to take action. Alarms are implemented through binary variables, 0 indicates no alarm and 1 indicates alarm. Figure 4.3(a) depicts the state of no-alarm and Figure 4.3 (b) depicts the state of alarm.



Figure 4.3: (a) State of no-alarm for Log No (b) State of alarm for PCSL LOGIC HEALTHY

An alarm is implemented through a disabled pushbutton and colour describes the state of the alarm. Yellow shows no-alarm and red shows alarm. Dynamic properties *TagName*

and *FlagVar* are used to identify an alarm in the screen. Both the properties have prefix "vb_" followed by the variable name of the alarm. For example, alarm *Log No* having variable name "vb_AN_01_1A" will be set to "vb_AN_01_1A" for both the properties. Identification of the alarms is discussed in section 4.8.

4.8.2 Pushbutton

Pushbuttons are used to give commands for opening the valves, moving/stopping a machine, starting/stopping a pump, etc. Pushbutton is implemented through binary variable, 1 indicates it is in pressed condition and 0 indicates it is in released condition. Pushbutton can be of momentary or latching type. Identification of the pushbuttons is discussed in section 4.10.

4.8.2.1 Momentary Type Pushbutton

Momentary pushbutton variable resets itself after it is released. It does not store the value and gets back to original position. Value of the variable is 1 until it is in pressed condition and 0 otherwise. "ALARM TEST" is one of the momentary pushbuttons. Figure 4.4(a) shows ALARM TEST in released condition and Figure 4.4(b) shows ALARM TEST in pressed condition.

Red colour depicts that the pushbutton is in pressed condition. Dynamic properties *TagName*, *VarName* and *FlagVar* properties are used. For example, for ALARM TEST pushbutton with variable name "PB_01_51_3_ALM", *VarName* is set to "pb_MOM_ALM_PNL1", *TagName* is set to "c_ PB_01_51_3_ALM " and *FlagVar* is set to " vb_PB_01_51_3_ALM ".



Figure 4.4: (a) ALARM TEST pushbutton in released condition. (b) ALARM TEST pushbutton in pressed condition.

4.8.2.2 Latching Type Pushbutton

Latching pushbutton variable latches (stores) the value until it is changed again. the conditions. Value of variable toggles with every time it is pressed and released. "EMERGENCY" pushbutton is one of the latching pushbuttons. Figure 4.5(a) shows the EMERGENCY pushbutton in non-latched or un-pressed condition and Figure 4.5 (b) shows the EMERGENCY pushbutton in latched condition.



Figure 4.5: (a) EMERGENCY pushbutton in released condition. (b) EMERGENCY pushbutton in pressed/latched condition

Red colour depicts that the pushbutton is in pressed condition. Dynamic properties *TagName*, *VarName* and *FlagVar* properties are used. For example, for EMERGENCY pushbutton having variable name "PB_EWS_C1_21_1", *VarName* is set to "pb_LATCH_SCR_SET", *TagName* is set to "c_PB_EWS_C1_21_1" and *FlagVar* is set to "vb_PB_EWS_C1_21_1".

4.8.3 Display Component

Display components are used to indicate values for important positions and parameters. Binary display components include LED, semaphore indicators and analog display components include 7-segment display, bar graph indicators. Identification of the display components is discussed in section 4.10.

4.8.3.1 LED

Light Emitting Diode (LED) is a display component that denotes a particular state of a device or specific position of the machine. These are used in different colours for different tasks. LED is represented by a binary variable in the simulator. Dynamic properties *TagName* and *FlagVar* are used to identify an LED in the screen. Both the properties take variable names with prefix "vb_" as LED variables are binary. An example of LED of various colours is shown in Figure 4.6.

LED for DSRDM -1 Locked with variable name "L_01_14_1_LCK" has *TagName* set to " vb_L_01_14_1_LCK" and *FlagVar* set to " vb_L_01_14_1_LCK". LEDs can be used in modules of one or more LED for a group of representations. Figure 4.7 (a-c) shows some of these modules.

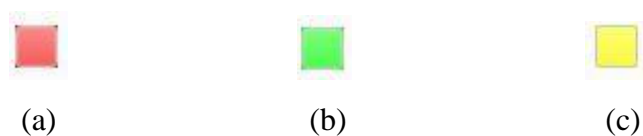


Figure 4.6: (a) Red color LED. (b) Green color LED. (c) Yellow color LED



(a)



(b)



(c)

Figure 4.7: (a) Two LED set for each of the three DSRDM (ENERGISED/DEENERGISED). (b) Two LED set for each of the three DSRDM (LOCKED/UNLOCKED). (c) Two LED set for each of nine CSRDM(ENERGISED/DEENERGISED)

4.8.3.2 Semaphore Indicator

Semaphore indicator is used in situations that require visual simulation of circuit or operating state of a device and indication of fault. It is implemented through binary variables, where one state represents value 1 and other state represents value 0. Figure 4.8(a) shows semaphore indicator with value 1 and Figure 4.8(b) shows semaphore indicator with value 0.



Figure 4.8: (a) Semaphore indicator with value 1. (b) Semaphore indicator with value 0

For example, Semaphore indicator with variable name "SI_12_3_2", has *TagName* set to "vb_SI_12_3_2" and *FlagVar* set to "vb_SI_12_3_2".

4.8.3.3 Digit Display

Digit display is a display element that display analog values. It is used for denoting positions and continuous values that are useful for the operator to analyse the conditions. It is represented by an analog variable in the simulator. Dynamic properties *TagName* and *FlagVar* are used to identify an analog display in the screen. Both properties have prefix "va_" as the seven segment take input an analog value. Figure 4.9 depicts a seven segment implementation in Qt.



Figure 4.9: Digit Display

For example, Seven Segment display with variable name "DYAPOWER", has *TagName* set to "va_DYAPOWER" and *FlagVar* set to "va_DYAPOWER".

4.8.3.4 Bar Graph Indicators

Bar graph indicators are used to show elevation in bar form. It is useful representation showing a pictorial way of displaying the elevation. These are used for displaying control rods elevation. It is represented by an analog variable in the simulator. Dynamic properties *TagName* and *FlagVar* are used to identify an analog display in the screen. Both properties have prefix "va_" as the seven segment take input an analog value. Figure 4.10 depicts a bar graph indicator implementation in Qt.

For example, Seven Segment display with variable name "BGI_01_8_1", has *TagName* set to "va_BGI_01_8_1" and *FlagVar* set to "va_BGI_01_8_1".

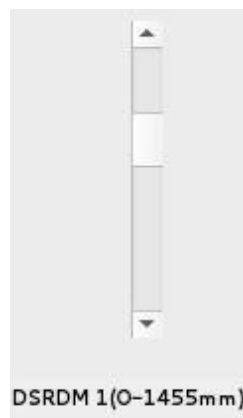


Figure 4.10: Bar Graph Indicator

4.8.4 Selector Switches

A selector switch (SS) is a device that is typically used to make selections between different positions. Qt does not provide any widget directly to implement selector switch. Thus, it is implemented through combination of pushbuttons and a label. Pushbuttons represent the different positions and label indicates the selection currently made. There are

three types of selector switches in simulator that are implemented, Normal SS, Key SS, Gang SS.

4.8.4.1 Normal Selector Switches

These are normal contact type selector switches in which selection is made by moving the selector by rotation without any authentication. Figure 4.11(a-f) shows the types of selector switches implemented.

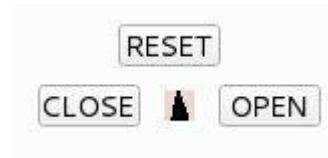
For example, a two position selector switch consists of one label and two pushbuttons. Clicking on the pushbutton is similar to rotating the switch of SS in hardware panel.

4.8.4.2 Key Selector Switches

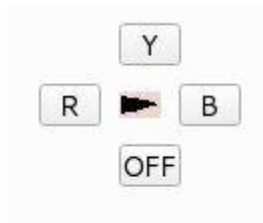
Key selector switch is similar to normal selector switch except that it requires an authenticated key to select a position. Figure 4.12 (a) shows different types of key selector switches implemented. A key selector switch is often combined with a two LED module where the selection on the switch is represented by LED. Figure 4.13 shows a key selector and two LED combinations.



(a)



(b)



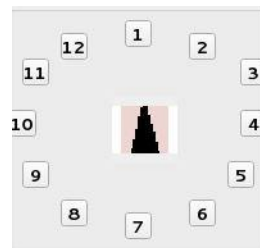
(c)



(d)



(e)



(f)

Figure 4.11: (a) Two Position SS. (b) Three Position SS. (c) Four Position SS
(d) Five Position SS. (e) Eight Position SS. (f) Twelve Position SS



(a)

Figure 4.12: (a) Two Position Key Selector Switch

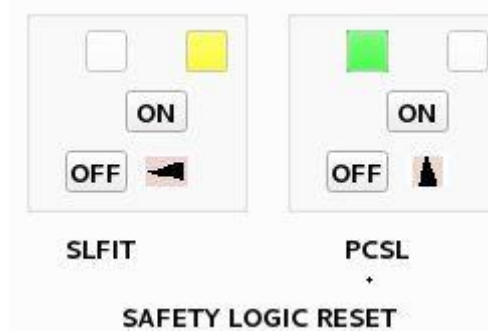


Figure 4.13: Key Selector SS and Two LED Combination

4.8.4.3 Gang Selector Switches

Gang switch are a series of latching type pushbuttons which function same as that of selector switch where only one selection can be made out of available choices. Figure 4.14 (a&b) shows the different type of gang selectors implemented.



(a)



(b)

Figure 4.14: (a) Gang SS for DSR Selection. (b) Gang SS for Power Range Selection

4.9 Parsing the User Interface (UI) and Storing Information

During loading phase of the virtual panels, component widgets in the panel are parsed and relevant information is stored. First, SQLite database is initialised and all the previous

stored information is erased from the database. DB3 database is also initialised as a requirement for simulator communication. Validity of the DB3 database is also checked against a template file. DB3 is an in-memory database. Due to its in-memory feature, it is faster than relational ex memory databases. Details of the files used for the DB3 database are mentioned in annexure 1. Then, all the widgets in the panels are parsed and a widget list is created. For each widget in the widget list, property *VarName* is read. If *VarName* is present, then property *TagName* is read otherwise that widget is skipped (means it is not a component). Each widget is individually identified and storage action is taken accordingly. The identification of widgets is described in Algorithm 4.1.

Algorithm 4.1 Identification of component widgets and storage of information

Input: List of component widgets for all the virtual panels (widgetlist)

Output: Store information of component in databases and assign an index

```

1: index=0
2: VP_CONF_REC *currBufPtr
3: for widget in widgetlist
4:     varname <- widget.property("VarName")
5:     if(varname!="")
6:         if(varname.startsWith("pb_"))
7:             tagname <- widget.property("TagName")
8:             len= tagname.length()
9:             if(tagname.startsWith("c_"))
10:                 varStatus <- CB
11:                 FillConfRec(widget, varStatus, len-2)
12:                 varStatus <- SB
13:                 FillConfRec(widget, varStatus, len-2)

```

```

14:             if(tagname.startsWith("f_"))
15:                 varStatus <- CB
16:                 FillConfRec(widget, varStatus, len-2)
17:         if(varname.startsWith("SS_") || varname.startsWith("KS_"))
18:             tagname <- widget.property("TagName")
19:             for widget2 in widgetlist
20:                 tagname2 <- widget2.property("TagName")
21:                 if(tagname2.contains(tagname)
22:                     len=tagname2.length
23:                     if(tagname2.startsWith("c")&&tagname2.at(2)=="_")
24:                         varStatus <- CB
25:                         FillConfRec(widget, varStatus, len-3)
26:                         varStatus <- SB
27:                         FillConfRec(widget, varStatus, len-3)
28:                     if(tagname2.startsWith("c")&&tagname2.at(3)=="_")
29:                         varStatus <- CB
30:                         FillConfRec(widget, varStatus, len-4)
31:                         varStatus <- SB
32:                         FillConfRec(widget, varStatus, len-4)
33:         if(varname.startsWith("GSPB_"))
34:             tagname <- widget.property("TagName")
35:             for widget2 in widgetlist
36:                 tagname2 <- widget2.property("TagName")
37:                 if(tagname2.contains(tagname)
38:                     len=tagname2.length
39:                     if(tagname2.startsWith("c")&&tagname2.at(2)=="_")

```

```

40:                                varStatus <- CB
41:                                FillConfRec(widget, varStatus, len-3)
42:                                varStatus <- SB
43:                                FillConfRec(widget, varStatus, len-3)
44:                                if(tagname2.startsWith("c")&&tagname2.at(3)=="_")
45:                                    varStatus <- CB
46:                                    FillConfRec(widget, varStatus, len-4)
47:                                    varStatus <- SB
48:                                    FillConfRec(widget, varStatus, len-4)
49:                                if(varname.startsWith("PB_C"))
50:                                    tagname <- widget.property("TagName")
51:                                for widget2 in widgetlist
52:                                    tagname2 <- widget2.property("TagName")
53:                                    if(tagname2.contains(tagname)
54:                                        len=tagname2.length
55:                                        if(tagname2.startsWith("c")&&tagname2.at(2)=="_")
56:                                            varStatus <- CB
57:                                            FillConfRec(widget, varStatus, len-3)
58:                                            varStatus <- SB
59:                                            FillConfRec(widget, varStatus, len-3)
60:                                        if(tagname2.startsWith("c")&&tagname2.at(3)=="_")
61:                                            varStatus <- CB
62:                                            FillConfRec(widget, varStatus, len-4)
63:                                            varStatus <- SB
64:                                            FillConfRec(widget, varStatus, len-4)
65:                                if(varname.startsWith("tg_"))

```

```

66:         tagname <- widget.property("TagName")
67:         for widget2 in widgetlist
68:             tagname2 <- widget2.property("TagName")
69:             if(tagname2.contains(tagname)
70:                 len=tagname2.length
71:                 if(tagname2.startsWith("cl_")
72:                     && tagname2. startsWith("cm_"))
73:                     && tagname2. startsWith("cr_")
74:                     varStatus <- CB
75:                     FillConfRec(widget, varStatus, len-3)
76:                     varStatus <- SB
77:                     FillConfRec(widget, varStatus, len-3)
78:         else
79:             if (tagname.startsWith("vb_"))
80:                 varStatus=SB
81:                 len=tagname.length()
82:                 FillConfRec(widget, varStatus, len-3)
83:             if (tagname.startsWith("va_"))
84:                 varStatus <- SA
85:                 len=tagname.length()
86:                 FillConfRec(widget, varStatus, len-3)
87:             if (tagname.startsWith("ca_"))
88:                 varStatus=CB
89:                 len <- tagname.length()
90:                 FillConfRec(widget, varStatus, len-3)

```

currBufPtr is an instance of VP_CONF_REC. It is a structure that has format as that of DB3 database. *FillConfRec* is the function that inserts data into both the databases. Algorithm 4.2 describes the insertion into the databases.

Algorithm 4.2 Insertion of data into databases

Input: widget, varstatus and offset

Output: Information stored in databases

```

1: VP_CONF_REC currBufPtr
2: currBufPtr.varNameGms <- widget.property("TagName")
3: currBufPtr.varFlagName <- widget.property("VarFlagName")
4: currBufPtr.varStatus <- varStatus
5: currBufPtr.varSysName <- widget.property("FlagVar").right(offset)
6: Insert a row into widgetdata(tagname,sysname,flagname,status,offset,offsetsize,index)
7: increment(index)

```

4.10 Pre-processing of the Information and Registering the Variables

Parsing of the virtual panels is done when change in the virtual panel is made. Otherwise, database once created will persist into the memory until modified by the program again. After parsing the widgets, information about the widgets is pre-processed to make it suitable for registering with the simulator and post interaction purposes. It is characterised by segregating all the information into different types (Pushbuttons, Selector Switches, Display Components) of data structures as discussed in section 4.5. The main purpose of the pre-processing is to arrange the components into the types so that whenever changes occurs,

minimal action is required to take the virtual panels to that state of simulator. Pre-processing reduces the post-work and arranges the components into sets of similar types. After pre-processing variables are registered with the simulator according to their status type. For example, when status type is SB/SA, then it is registered for subscription as a binary/analog variable. When status type is CB/CA, then it is registered for publishing as a binary/analog variable. Algorithm 4.3 shows the steps involved in pre-processing and registering the variables.

For registering the variables, functions described in section 2.2.1.2.3 are used. External programs use the IPC wrapper which abstracts the use of *MDSM* functions. All the variables are registered based on their status. To register a variable for publishing (varStatus = CB/CA) *IPC_RegVarByVal* is used.

```
IPC_RegVarByVal(modVarsInfo[i].varNameSys, &modVarsInfo[i].Value.binValue,
               sizeof(WORD))
```

```
IPC_RegVarByVal(modVarsInfo[i].varNameSys,
               &modVarsInfo[i].Value.analogValue, sizeof(float))
```

It takes three arguments, name of the variable *varNameSys*, address of the local memory containing the value to be published and size of the address field. For a binary variable size of WORD is used and *binvalue* field of *modVarsInfo* structure is used. For an analog variable, size of float is used and *analogValue* field of *modVarsInfo* is used. Variable *i* refers to the index of that variable assigned during the parsing.

To register a variable for subscription (varStatus = SB/SA), *IPC_SubVarByVal* is used.

```

IPC_SubVarByVal(modVarsInfo[i].varNameSys,
                &modVarsInfo[i].Value.binValue, sizeof(WORD));

IPC_SubVarByVal(modVarsInfo[i].varNameSys,
                &modVarsInfo[i].Value.analogValue, sizeof(float));

```

It takes three arguments name of the variable *varNameSys*, address of the local memory where simulator should write the value of the variable and size of the address field. For a binary variable, size of WORD is used and *binvalue* field of *modVarsInfo* structure is used. For an analog variable, size of float is used and *analogValue* field of *modVarsInfo* is used. Variable *i* refers to the particular index of that variable assigned during the parsing.

Identification of the components is done in similar way as performed during the parsing. The data from the previous stage when it was stored in DB3 is registered with the simulator using the status of the variable. Now the simulator uses indexed position of *modVarsInfo* and the virtual panels will use index from the SQLite database. This way synchronisation is done for accessing the variables. Algorithm 4.3 gives a high level description of segregation of component widgets.

Algorithm 4.3 Segregation and Storage of Component widgets

Input: widgetlist

Output: Information of components segregated into types

- 1: PBS pb
- 2: SSS ss
- 3: TGS tg

```

4: for widget in widgetlist
5:     varname <- widget.property("VarName")
6:     if(varname!="")
7:         tagname <- widget.property("TagName")
8:         if(varname.startsWith("pb_"))
9:             index <- Find the index of tagname in SQLite widgetdata table
10:            if(tagname.startsWith("c_"))
11:                Store information in pb
12:            if(varname.startsWith("pb_MOM"))
13:                pb.type= MOM
14:            else if(varname.startsWith("pb_LATCH"))
15:                pb.type= LATCH
16:            if(varname.startsWith("SS_") || varname.startsWith("KS_"))
17:                index <- Find the index of tagname in SQLite widgetdata table
18:                tagname <- widget.property("TagName")
19:                for widget2 in widgetlist
20:                    tagname2 <- widget2.property("TagName")
21:                    if(tagname2.contains(tagname))
22:                        len=tagname2.length
23:                        if(tagname2.startsWith("c")&&tagname2.at(2)=="_")
24:                            Store information in ss
25:                        if(tagname2.startsWith("c")&&tagname2.at(3)=="_")
26:                            Store information in ss
27:                    if(varname.startsWith("GSPB_"))
28:                        tagname <- widget.property("TagName")
29:                        index <- Find the index of tagname in SQLite widgetdata table

```

```

30:         for widget2 in widgetlist
31:             tagname2 <- widget2.property("TagName")
32:             if(tagname2.contains(tagname)
33:                 if(tagname2.startsWith("c")&&tagname2.at(2)=="_")
34:                     Store information in ss
35:                 if(tagname2.startsWith("c")&&tagname2.at(3)=="_")
36:                     Store information in ss
37:         if(varname.startsWith("PB_C"))
38:             tagname <- widget.property("TagName")
39:             index <- Find the index of tagname in SQLite widgetdata table
40:             for widget2 in widgetlist
41:                 tagname2 <- widget2.property("TagName")
42:                 if(tagname2.contains(tagname)
43:                     if(tagname2.startsWith("c")&&tagname2.at(2)=="_")
44:                         Store information in ss
45:                     if(tagname2.startsWith("c")&&tagname2.at(3)=="_")
46:                         Store information in ss
47:         if(varname.startsWith("tg_SS"))
48:             tagname <- widget.property("TagName")
49:             for widget2 in widgetlist
50:                 tagname2 <- widget2.property("TagName")
51:                 if(tagname2.contains(tagname)
52:                     if(tagname2.startsWith("cl_")
53:                         && tagname2.startsWith("cm_"))
54:                         && tagname2.startsWith("cr_")
55:                     Store the information in ss

```

```

56:
57:         else if (tagname.startsWith("vb_"))
58:             index<- Find the index of tagname in SQLite widgetdata table
59:             Store the information in binaryvarpd
60:         else if (tagname.startsWith("va_"))
61:             index<- Find the index of tagname in SQLite widgetdata table
62:             Store the information in analogvarpd
63:         else if (tagname.startsWith("ca_"))
64:             index <- Find the index of tagname in SQLite widgetdata table
65:             Store the information in Commanalogvarpd

```

4.11 Interaction with User Interface (UI)

Components of the panel are interactive depending on the action performed. State of the component may or may not change depending on the type of the component. Components and their actions upon interaction is described in further sections.

4.11.1 Pushbutton

A pushbutton has two actions, PRESS and RELEASE. When a pushbutton is pressed/released, Qt generates a signal (function) which can be implemented and actions that need to be done after the PRESS/RELEASE must be written in the SLOT function of the action. For a momentary pushbutton, both the actions are defined but for a latching type pushbutton, only PRESS action is defined. Latching type toggles between the values when pressed. If current value is 0 then, in next PRESS event it will change to 1 and vice-versa.

When a pushbutton is pressed, its index is searched into the SQLite database using the *tagname* read from the dynamic property of the pushbutton. Then the event (PRESS/RELEASE) is stored into a data structure of pushbutton and flags for publishing the data are set. When flags for publishing are checked, variables are published from the data stored in the buffer. Pseudocode 4.1 describes the actions taken when a pushbutton pressed action is initiated.

Pseudocode 4.1 Actions when a Pushbutton is Pressed/Released

Input: None

Output: Pushbutton action triggered, event recorded and publish flag set

- 1: Find the tagname of the pushbutton.
 - 2: Search for the index in SQLite table.
 - 3: Pass the information to the database along with the index found
 - 4: Store the Event.
 - 5: Set Publish Flag and return
-

4.11.2 Selector Switch

Selector switch is implemented in the similar way logically for all three (normal, key selector, gang selector) types. SS data structure is used for all types. In selector switch, one of the options is selected at a time. If an option is marked selected then, variable corresponding to it represents 1 and other options have value 0. Selector switch is implemented through combination of pushbuttons and label. Pressing a pushbutton generates an action that the selection is being made. After a pushbutton on the selector switch is pressed (event= PRESS),

action is generated and *TagName* of the pushbutton is read. After reading the *TagName*, index corresponding to the *TagName* is searched in the SQLite table. Then data is saved into the buffer and flags are set for publishing. Pseudocode 4.2 describes actions taken when selector switch is interacted upon.

Pseudocode 4.2 Actions when a Selector Switch Pushbutton is Pressed/Released

Input: None

Output: Selector Switch action triggered, event recorded and publish flag set

- 1: Find the tagname of the pushbutton.
- 2: Search for the comment in SQLite table.
- 3: Pass the information to the database along with the index found
- 4: Search which SS is Pressed and Store the Event.
- 5: Set Publish Flag and return

4.11.3 Toggle Switch

Toggle switch is similar to the selector switch. The difference is that it returns back to its original position when released. It means that variable is 1 until the selection is made by force. In the implementation, until the pushbutton is pressed value of the variable of that pushbutton variable is set 1 and resets after it is released. . After a pushbutton on the toggle switch is pressed (event= PRESS), action is generated and *TagName* of the pushbutton is read. After reading the *TagName*, index corresponding to the *TagName* is searched in the SQLite table. Then data is saved into the buffer and flags are set for publishing. Pseudocode 4.3 describes actions taken when selector switch is interacted upon.

Pseudocode 4.3 Actions when a Toggle Switch is Pressed/Released

Input: None

Output: Selector Switch action triggered, event recorded and publish flag set

- 1: Find the tagname of the pushbutton.
 - 2: Search for the comment in SQLite table.
 - 3: if(event = PRESS)
 - 6: Pass the information(pressed) to the database along with the index found
 - 4: Search which SS is Pressed and Store the Event
 - 5: else if(event = RELEASE)
 - 7: Pass the information(pressed) to the database along with the index found
 - 6: Read Prev Property of the widget
 - 7: Search the index of with Prev
 - 8: Pass the information(pressed) to the database along with the index found
 - 8: Set Publish Flag and return
-

4.12 Updating the User Interface (UI)

User interface is updated each cycle after the variables are read from simulator. User gets all the information about the system from the user interface. Initially the user interface shows default values until the simulator is not running. When the simulator starts running, all the values are updated for the current screen only reducing the load on the process and simulator. For updating the screens, the variables are registered as described in algorithm 4.3 and values are used accordingly. For example, LED glows when value of the variable is 1 and does not glow when value of the variable is 0. Algorithm 4.7 describes the values updation,

and how each component is differentiated in order to display correct values. *startsWith()* is the function that checks if the string starts with the string specified in the arguments.

Algorithm 4.4 Updating the virtual panel and component widgets

Input: widgetlist

Output: Updated virtual panel

```
1: Open the database SQLite with widgetdata table
2: for each widget in widgetlist
3:     tag <- widget.property("TagName")
4:     varname <- widget.property("VarName")
5:     if(tag.startsWith("vb_"))
6:         find index of tag in widgetdata
7:         if(tag.startsWith("vb_AN_"))
8:             Update Alarm Color using modvarsinfo[index]
9:         if(tag.startsWith("vb_L_"))
10:            Update LED colour using modvarsinfo[index]
11:        if(tag.startsWith("vb_SI"))
12:            Update Semaphore Indicator using modvarsinfo[index]
13:    if(tag.startsWith("c_"))
14:        find index of tag in widgetdata
15:        if(widget is PushButton)
16:            Update Pushbutton using modvarsinfo[index]
17:    if(tag.startsWith("ss2"))
18:        find index of tag in widgetdata for all pushbuttons in SS
19:        Set Arrow using modvarsinfo[index]
20:    if(tag.startsWith("ss3"))
```

```

21:         find index of tag in widgetdata for all pushbuttons in SS
22:         Set Arrow using modvarsinfo[index]
23:     if(tag.startsWith("ss4"))
24:         find index of tag in widgetdata for all pushbuttons in SS
25:         Set Arrow using modvarsinfo[index]
26:     if(tag.startsWith("ss5"))
27:         find index of tag in widgetdata for all pushbuttons in SS
28:         Set Arrow using modvarsinfo[index]
29:     if(tag.startsWith("ss8"))
30:         find index of tag in widgetdata for all pushbuttons in SS
31:         Set Arrow using modvarsinfo[index]
32:     if(tag.startsWith("ss10"))
33:         find index of tag in widgetdata for all pushbuttons in SS
34:         Set Arrow using modvarsinfo[index]
35:     if(tag.startsWith("ss12"))
36:         find index of tag in widgetdata for all pushbuttons in SS
37:         Set Arrow using modvarsinfo[index]
38:     if(tag.startsWith("va_"))
39:         if(widget is Pushbutton)
40:             widget.setText()
41:         if(widget is Label)
42:             widget.setText()
43:         if(widget is Dial)
44:             widget.setdial()
45:         if(widget is Scrollbar)
46:             widget.setscrollbar()

```

```
47: if(varname.startsWith("GSPB"))  
48:     find all the index related to this widget  
49:     Update the pushbutton using modvarsinfo[index]
```

4.13 Results of the Development

After implementation of all the components, components are consolidated in the screens according to the panel to which they belong. Initially when the simulator is not in RUN mode, all the components show the default value set for them during the registering. Figure 4.15, Figure 4.16 and Figure 4.17 show the initial behaviour of the alarm tab, display tab and controls tab of the panel 1 respectively. Figure 4.15 includes all the alarms in the panel 1. Yellow color represents that either alarm is having value 0. Red color represents the alarm state. Figure 4.16 depicts the display tab which includes many display components and some control components also. It includes bar graph indicator, LEDs, selector switches, and many combinations of LEDs which are present in panel 1. Figure 4.17 depicts the controls tab which shows all the inhibition key selector switches in panel 1 and control components in the panel 1. Components and virtual panels depicted in the chapter are not integrated with the simulator to display the real time data. Integration is explained in the next chapter. Developed virtual panels are raw panels with some information with them which is worthless unless processed by the framework and given a meaningful form through the real time changes in the virtual panel components.

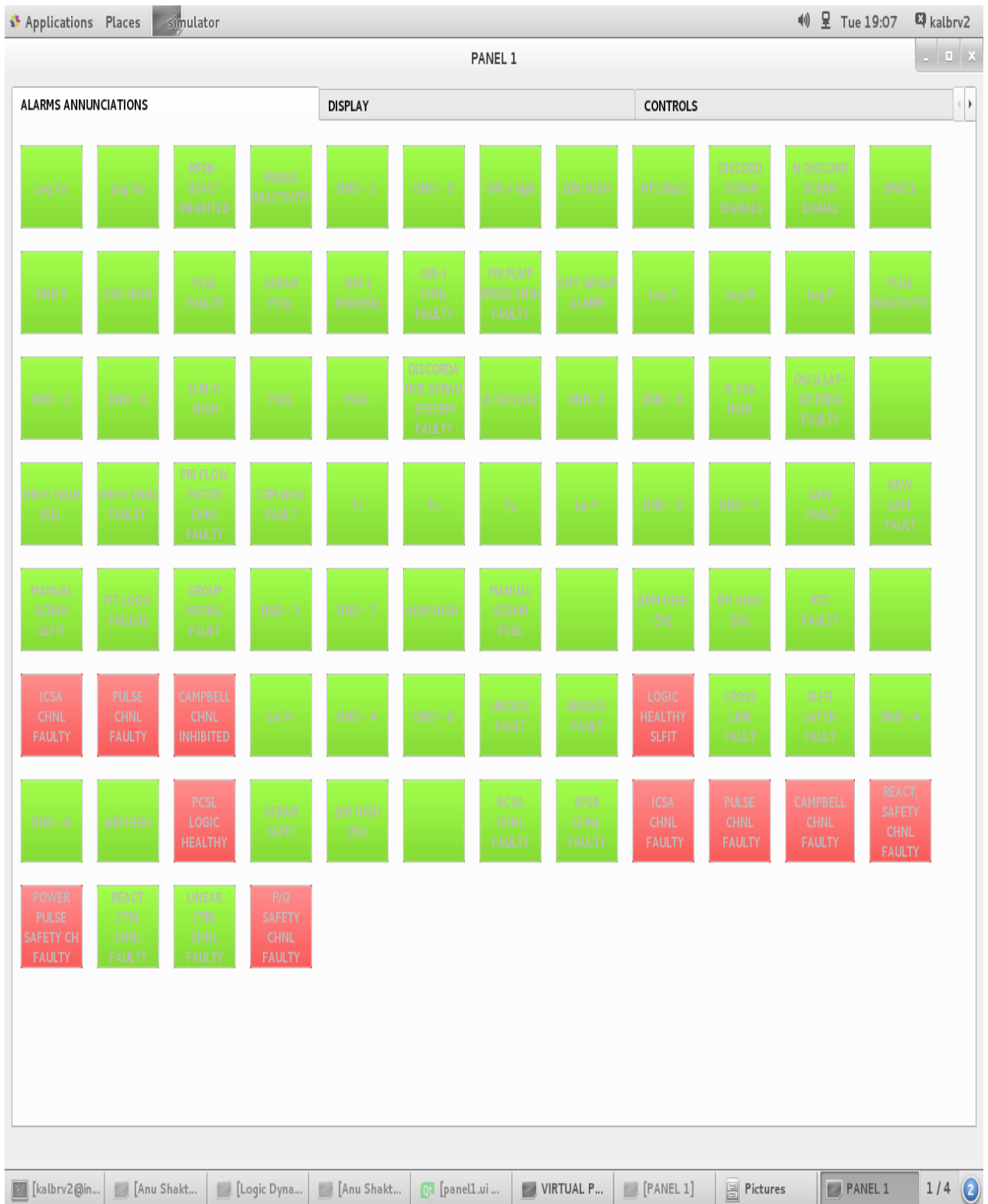


Figure 4.15: Alarm tab of Panel 1

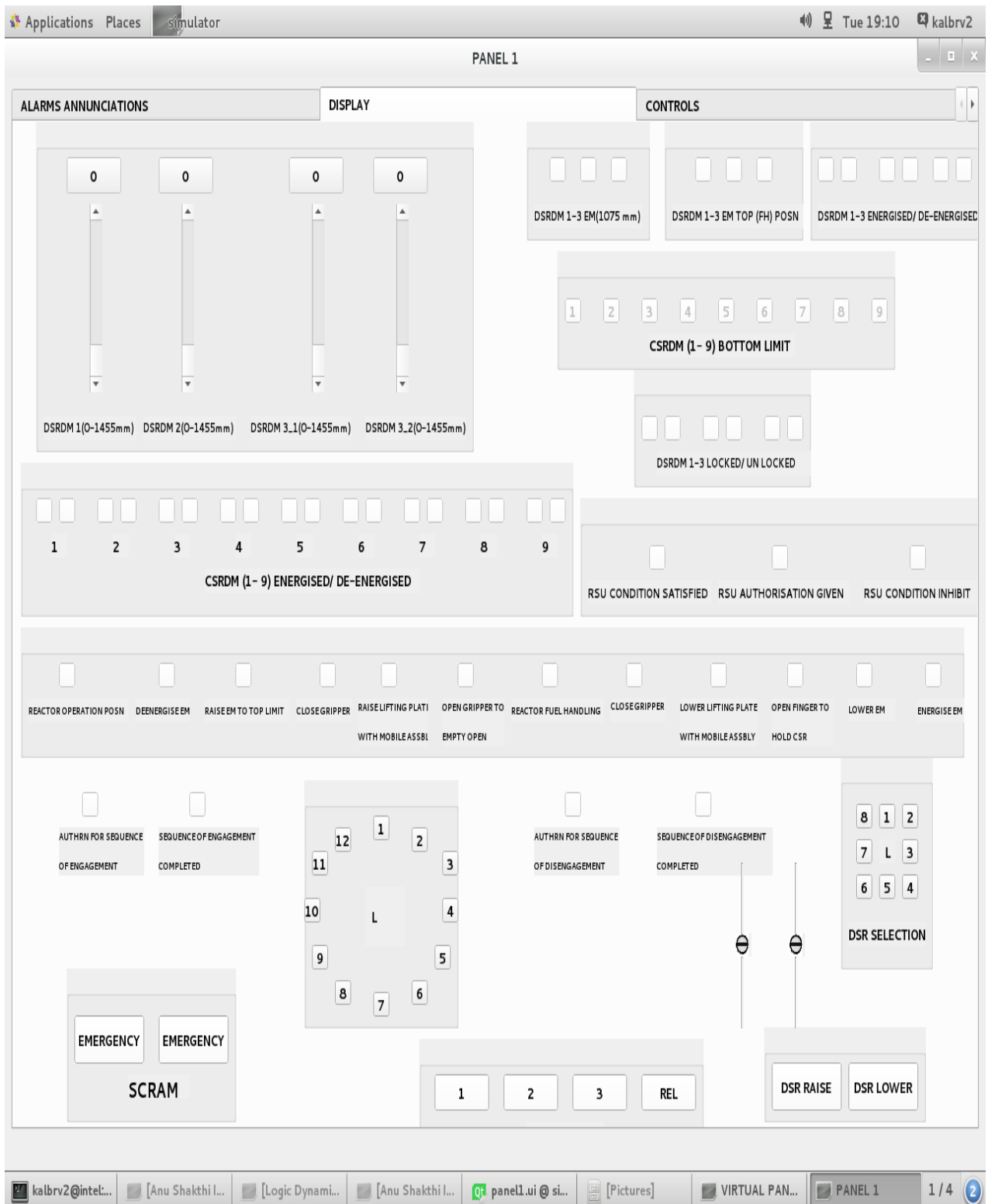


Figure 4.16: Display tab of Panel 1

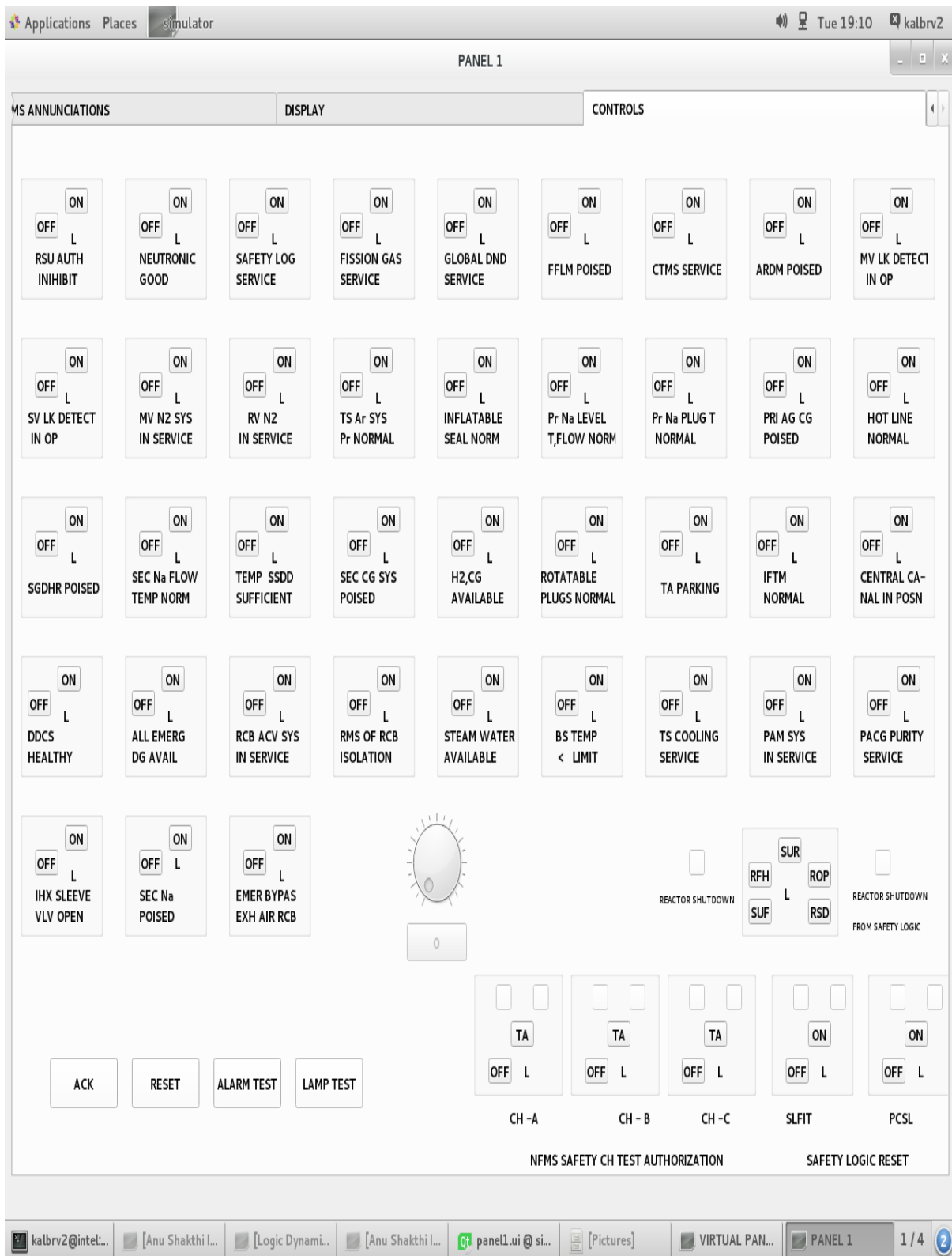


Figure 4.17: Controls tab of Panel 1

Chapter 5

Integration of Virtual Panels with the Simulator

Integration of the developed components forms the final and an important part of the project. Integration refers sharing of simulator data with virtual panels and interaction of the components. Simulator database is accessed and modified for all the interactions taking place with the virtual panels. This chapter discusses about the integration mechanism of the process with the daemon processes. Integration in its core refers to registering the process and interacting/sharing the messages with the simulator processes. De-Registering of the process after the lifetime of the process is also discussed. When lifetime of a process ends, it must perform activities so that next time when it joins, state of process is not changed. In terms of simulator it is termed as saving the state of the simulator.

5.1 IPC Implementation and Registering Process with Simulator

Functions mentioned in section 2.2.1.2 are implemented through IPC wrapper which abstracts the implementation and internal details. These Functions are used through the wrapper and no function of MDSM is used directly. It simplifies the call for the programmer.

The details of the synchronous mechanism and messaging protocols can be abstracted from user.

Function *IPC_GuiInit* is used to register a GUI process with the simulator.

```
int IPC_GuiInit( char *proc_name,
                char *directory,
                int publishflag,
                int timeout,
                void ( *Reg_fn )( void ),
                void ( *Subsc_fn )( void ),
                void ( *MsgRec_fn )( void *data, int MsgType, int size, MDSM_ProcId *sender)\,
                int ( *SimStateSave_fn )( char *simStateId, int simStateType),
                int ( *SimStateLoad_fn )( char *simStateId, int simStateType),
                void ( *ReplayCBFn)(char *data,int len),
                void ( *CleanUpFn)(int signo),
                void ( *SimStateDelete_fn)( char *simStateId, int simStateType),
                void ( *PanelAlign_fn)( int operationFlag ),
                void ( *IoTest_fn)( int operationFlag, char *sender, void *data ) )
```

This function performs the necessary initializations required for Inter Process Communication(IPC). It registers the process with the MDSM, and sets up table of callback functions for the various types of message events described below. This function is NON-BLOCKING meaning it does not wait for the response for the return value and lets the program to continue until value is returned.

The arguments of the function are explained below:

proc_name : The name of the process registering with IPC routines.

directory : The file with the MDSM configuration information. This file tells the IPC routines which socket number to use and upon which machine the messaging is located.

publishFlag : If set to one, then the executive expects this process to publish variables. If set to zero, the executive does not wait for this process to publish.

timeout : This tells executive to send the run signal to this process every "timeout" time steps. In case of GUIs, it is not critical that the process receive signal every cycle time. It can be set to every 2nd or 3rd cycle of the models. This is a tuning parameter for the timing of the simulator.

Reg_fn : A callback function in which initial registration of variables is done. This is to prevent subscription before publishing.

Subsc_fn : A callback function in which all of the subscription are done. The executive waits for all the processes to finish the *Reg_fn* call before ordering the subscription of variables.

MsgRec_fn : A callback function used when *IPC_PollMsg()* [Explained in section 5.2] function is called, and a message is received.

SimStateSave_fn : This callback function will be called when it is time to save the simulator state. The parameter *SimStateId* identifies the file simulator is going to be saved in. *simStateType* identifies the type of the simulator save, IC, Backtrack, Snapshot.

SimStateLoad_fn : This callback function will be called when it is time to load the simulator state. The parameters are similar to *SimStateSave_fn*.

CleanUpFn : This callback function is used when the process exits the simulator system.

SimStateDelete_fn : This callback function notifies the process that an IC, Snapshot, or Backtrack has been deleted.

Return : IPC_OK for success, IPC_ERR for failure, IPC_NODAEMON if messaging daemon is not detected.

5.2 Communication with the Executive

Inter process communication between the executive and the process takes through messages. Whenever a message is received from the executive, it is checked against the options and action is taken accordingly. If the message is IPC_START, then all the values which are subscribed by the process are copied to the local memory of the process. If the message is IPC_PUBLISH, then all the variables registered are published. Alternatively, only modified variables can be published to reduce the redundant load on simulator. If no message is received then keep polling for new messages. Function used for polling the messages is *IPC_MsgPoll*.

void IPC_MsgPoll(void)

This function polls for messages from other processes. If the message is forthcoming, the user defined function *MsgRec_fn()* is called with the data packet and the data type.

5.3 Sending User Interface (UI) Responses to Simulator

In section 4.10, interaction with UI was discussed. After interacting with UI, a signal from UI is generated which prompts call to a function. Function finds out the index of the variable of component widget. Then using the data structure saved for the widget combinations, appropriate values are set for the widget components. Widgets are updated after the values are published and sent back the process through subscription mechanism. If a pushbutton is pressed then event PRESS is sent and appropriate function is called and values for the variables are set to one. In case of a selector switch, the pushbutton PRESSED is set to one and all other options in the selector switch are set to zero. Along with setting the values in the data structure of selector switch, flag are also set for publishing. When the executive sends IPC_PUBLISH, the process publishes all the registered variables using *IPC_PublishAll()*.

5.4 Saving the state of User Interface (UI) and De-Registering the Process and Variables

When a process leaves the simulator system, it should save the current state of simulator so that if it returns back it should find the simulator in the same state as when it left. All the variables which are published by the process are saved in the same database as they were stored to interact with the simulator. While leaving the system, a process should de-register itself and all the variables it has registered with the MDSM. Functions for de-registering process and variables are explained in the sections 2.2.1.2.8 and 2.2.1.2.3.3 respectively.

5.5 Results of Integration

Section 4.12 shows the development results for panel 1 and console 1. Results shown in section 4.12 are incomplete in the sense that they show default values rather than values from the simulator database. Default values are displayed when simulator has not started. After integration with simulator is complete and simulator is in RUN state, they show the correct simulator values. Figure 5.1, Figure 5.2 and Figure 5.3 show the status of the virtual panels after the integration for alarm, display and controls tab of panel 1.

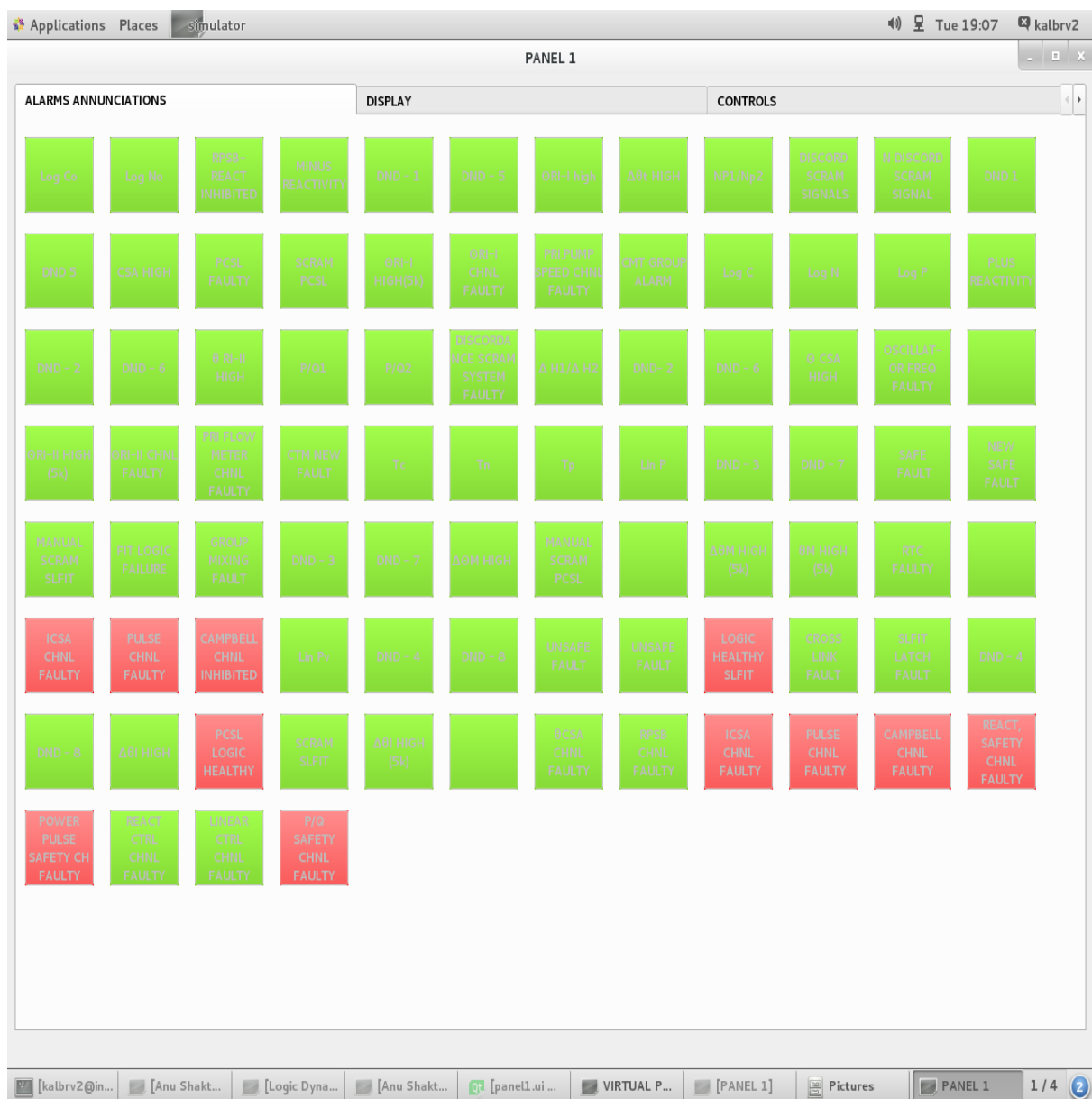


Figure 5.1: ALARM tab of Panel 1 after integrating with simulator

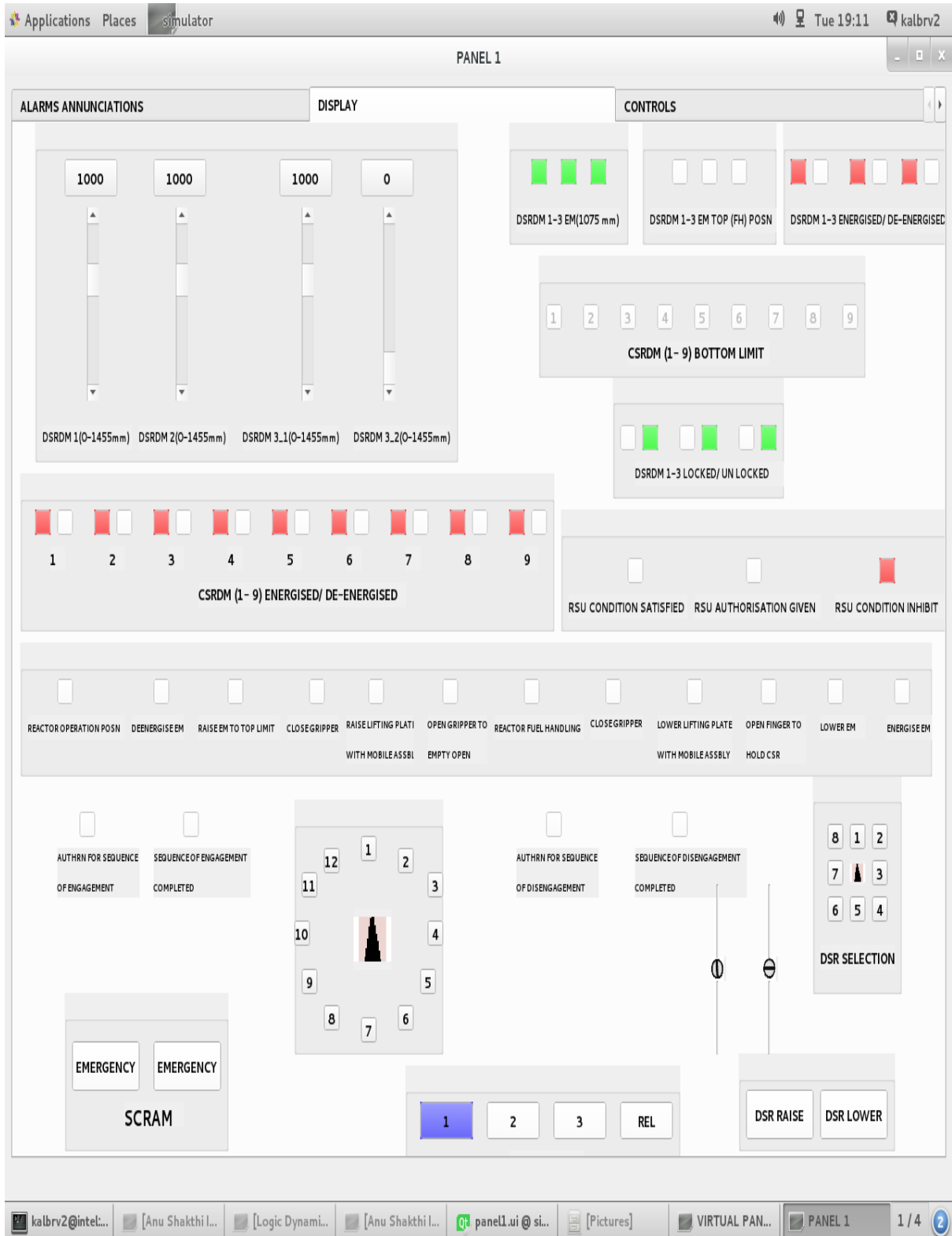


Figure 5.2: DISPLAY tab of Panel 1 after integrating with simulator

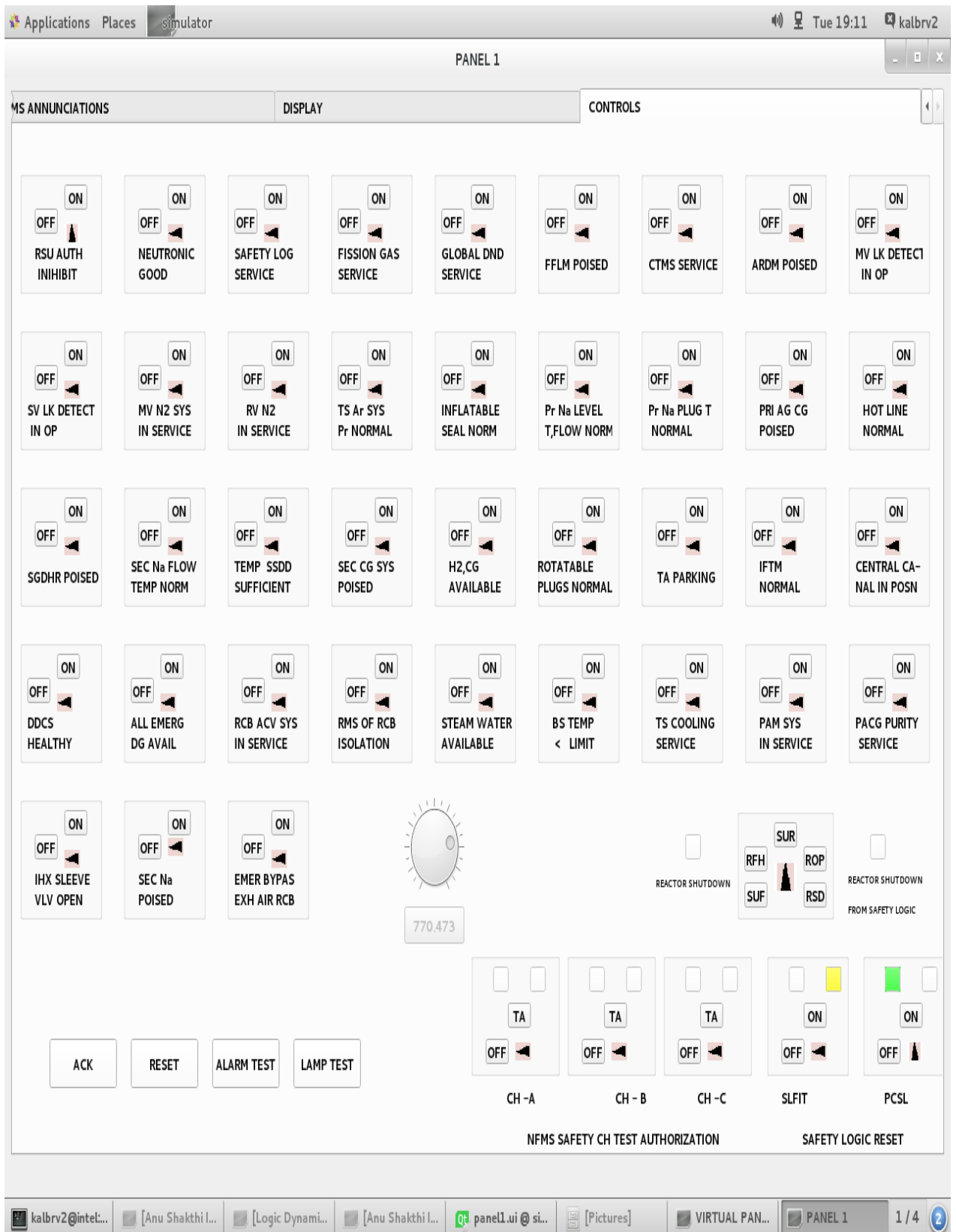


Figure 5.3: CONTROL tab of Panel 1 after integrating with simulator

Chapter 6

The Methodology of reusing the Framework

This chapter explains how template widgets developed can be reused to create more virtual panels. It is implemented using the properties defined for the each widget and the code for identifying and using the properties of components. Framework reduces efforts for creating virtual panels through reusability.

6.1 Creating a virtual panel

Creating a virtual panel and integrating it with the simulator is done using the graphical user interface of the development software (Qt Creator). Each virtual panel has two aspects to be taken care, one being the user interface model file (.ui file) and the other is the associated C/C++ code to drive the panel. As most of the panels have similar components/widgets contained in them, creating new panels is easily achieved. It involves replicating and renaming the user interface file and associated code to match the new panel.

For example, the Alarms of all the virtual panels are similar except the names of alarms and their concerned simulator variables. Taking advantage of this any existing panel say, alarms_panel_1.ui can just be copied and made as alarms_panel_2.ui and the names of

the alarms alone can be changed as required. Similarly the code/class file associated with alarm_panel_1.cpp can be copied and renamed to match alarm_panel_2.cpp. Once the code and the user interface files are ready they can be compiled and integrated with the simulator. Any new component/widgets can be added or removed by copying and pasting from other virtual panels or deleting from this panel as needed. Next section describes how components are added to the virtual panel.

6.2 Incorporating Components in the Virtual Panel

After creating a virtual panel, components are added to it. Each component already created and integrated with the simulator is copied and pasted to the required virtual panel from the source panel where it exists or from template virtual panel. As each component in each panel has a different variable, so variable of the created component is changed to suit the functionality. Names of the variable can be referred from the database available in the simulator room. Nomenclature of the components and their properties is explained in the chapter 4.

Framework is collection of the code that manages the component widgets on the virtual panel and performs the tasks for generic widgets developed. Framework code identifies the new widgets and stores them in the database which finally helps in interacting with the simulator daemon processes. Each of the virtual panel contain three tabs namely, annunciation, display and controls which are being shown and explained in section 4.12 and 5.5. All the pre-processing, identification and interactions are taken care by the program itself if the nomenclature is followed accordingly.

To add the component, open the virtual panel, where the component to be created exists. Select the component that is to be created. Copy the component by selecting it. Open

the panel where component needs to be copied and paste it. Place the component to a relevant location on the panel. Change the dynamic properties of the component referring the database and save it. Properties which are to be changed can be found out from section 4.7. Recompile and run the program.

Above procedure is for all the components and user can decide the placement of the component. Any number of the components can be fitted in the virtual panel as long as the panel doesn't look cluttered. Name of the component should be changed and checked. Behaviour of component remains same as parent component unless variable is changed.

Chapter 7

Summary and Future Work

Simulators play a very important role in training of operators. It gives a feeling to the operator as if working in a real control room. All functionalities, even the hardware and non-nuclear aspects should be same so as to extract maximum output in behavioural and technical aspects. Simulator consists of many models integrated together. Virtual Panel is one of the models. Process and logic models are ported to Intel platform. This project dealt with the third important model to be developed for Intel platform. Development of framework for creating virtual panels is the objective of the project. Framework for creating virtual panels for the Intel simulator is developed using Qt and tested by developing three panels. More number of virtual panels can be added using the development Integrated Development Environment (IDE) QtCreator. Component widgets can be added to the virtual panel using the developed widgets. All the components were developed and tested for interaction with simulator. Behaviour of each of the component developed is observed and analysed. Example of each type of widget is depicted in relevant chapters. Algorithm for each of the tasks is described in the relevant chapters.

7.1 Future Work

Presently, virtual panels are developed for three hardware panels with components. There are more than 21 hardware panels in the simulator which can be developed using the framework developed for creating the virtual panels. Components in the created virtual panels can be taken from the developed virtual panels or the template virtual panel. More components can be added to the project for further expansion. Further, modification of the components through changing properties can be done. Colours and size of the components can be modified as required.

Annexure A

A.1 Database Files used for DB3 setup

dBase[48] is one of the first database management systems for microcomputers, and most successful in its days. *dBase*'s underlying file format, the .dbf is used in applications where simple format is required to store structured data. It is a file based storage type database. It is copied to memory when required and saves lot of disk read and writes. Virtual panels use this database to store the information required to communicate with the simulator daemon processes. Format of the storage information is specified in section 4.5. Each cycle of scan of virtual panel screens leads to resetting the databases. As this is a file storage type database, everything is erased when the file is cleared. Thus a template file is required which would store the first record of columns. Although accessing mechanism for the database is sequential, file is read/write record by record to implement the relational essence. The first file is a template file which needs to be included so that when the database is cleared, again template file can be first written to the main database file and each record can be checked against the template to check its correctness. This is how the database correctness is checked. When the program starts, whole file is read into the memory record by record checking the record size and type against the template header. In case of a mismatch, error code is returned and program exits.

A.2 Types of Widget Component and Actions with Codes

Each type of the widget component is identified by the type of code its data structure contains. Code is assigned during the scanning of the screens and identification of the components. Code is assigned only to the interactive components as they only need to be identified during the interaction. Non interactive components can be directly identified using their prefixes assigned by the user and are identified as explained in the algorithms in chapter 5. Codes for various components and actions are given below:

TOGGLE SWITCH	TGS	1
PUSHBUTTON	PB	2
SELECTOR SWITCH WITH 2 PUSHBUTTONS	SS2	3
SELECTOR SWITCH WITH 3 PUSHBUTTONS	SS3	4
SELECTOR SWITCH WITH 4 PUSHBUTTONS	SS4	5
SELECTOR SWITCH WITH 5 PUSHBUTTONS	SS5	6
SELECTOR SWITCH WITH 8 PUSHBUTTONS	SS8	7
SELECTOR SWITCH WITH 10 PUSHBUTTONS	SS10	8
SELECTOR SWITCH WITH 12 PUSHBUTTONS	SS12	9
LATCHING PUSHBUTTON	LATCH	30
MOMENTARY PUSHBUTTON	MOM	31
PRESS ACTION	PRESS	0
RELEASE ACTION	RELEASE	1

Annexure B

B.1 Pseudocode for the Algorithms

Pseudocode [51] is an artificial and informal language that helps programmers to develop algorithms. Pseudocode is a "text-based" detail (algorithmic) design tool. The rules of Pseudocode are reasonably straightforward. All statements showing "dependency" are to be indented. These include while, do, for, if, switch. Pseudocode for algorithms in chapter 4 are discussed below.

B.1.1 Pseudocode for Algorithm 4.1

##4.4: Identification of components widgets and storage of information

Set index to zero

Initialise the current buffer pointer

for all the widgets in the widget list

 Read variable name property of the widget

 If variable name is present for the widget then

 If the variable name has prefix of pushbutton

 Read the Tag Name property of the widget

 Read the length of the tag name

 If tag name has prefix for publishing then

 Store the widget information for publishing

Store the widget information for subscribing

If tag name has prefix for flag then

Store the widget information for publishing

If tag name has prefix for selector switch or key selector switch then

Read the tag name property of the widget

for all the widget in widget list

Read the tag name property of the widget

Read the length of the tag name

If tag name has prefix for publishing and tag name has numbering

Store the widget information for publishing

Store the widget information for subscribing

If tag name has prefix for gang selector switch or toggle switch then

Read the tag name property of the widget

for all the widget in widget list

Read the tag name property of the widget

Read the length of the tag name

If tag name has prefix for publishing and tag name has numbering

Store the widget information for publishing

Store the widget information for subscribing

Else

Read the tag name property of the widget

Read the length of the tag name

If tag name has prefix for input binary widget then

Store the widget information for subscribing binary

If tag name has prefix for input analog widget then

Store the widget information for subscribing analog
If tag name has prefix output analog publishing then
Store the widget information for publishing analog

B.1.2 Pseudocode for Algorithm 4.2

##4.5: Insertion of data into the databases

Declare current buffer pointer for the panel
Read the tagname, flagname, status and length of the flagname
Insert a row into the initialised database with index number and tagname as the primary key
Increment the index
exit

B.1.3 Pseudocode for Algorithm 4.3

##4.6: Segregation and Storage of component widgets

Declare instances of pushbutton, toggle switch and selector switch structures
For all the widgets in the widget list
 Read variable name property of the widget
 If variable name is present for the widget
 Read the tag name property of the widget
 If variable name has prefix for pushbutton
 Find the index in the table using key tagname
 If tag name has prefix for publishing
 Fill the information in pushbutton structure

If variable name has prefix for momentary type

Set type of button to be momentary type

If variable name has prefix for latching type

Set type of button to be latching type

If the variable name has prefix for selector switch or key selector switch

Find the index in the table using key tagname

Read the tagname of the widget

For all the widgets in the widget list

Read tag name of the widget

If tag name contains tag name of the selector switch or key selector switch tag name

Fill the information of widget in selector switch structure

If the variable name has prefix for gang selector switch

Find the index in the table using key tagname

Read the tagname of the widget

For all the widgets in the widget list

Read tag name of the widget

If tag name contains tag name of the selector switch or key selector switch tag name

Fill the information of widget in selector switch structure

If the variable name has prefix for toggle type selector switch

Find the index in the table using key tagname

Read the tagname of the widget

For all the widgets in the widget list

Read tag name of the widget

If tag name contains tag name of the selector switch or key selector switch tag name

Fill the information of widget in selector switch structure

If tagname has prefix for digital input and outputs

Find the index in the table using key tagname

Fill the information of widget in corresponding binary, analog structure

References

- [1] "Developing a Systematic Education and Training Approach Using Personal Computer Based Simulators for Nuclear Power Programmes", *IAEA - TECDOC - 1836*, May 2017.
- [2] P Chellapandi et al., "Fast Reactor Programme in India", *Indian Academy of Sciences*, Vol. 85, No. 3, September, 2015.
- [3] T. Jayanthi, S.A.V. Satyamurthy, P. Swaminathan, "Training simulators and their role in Nuclear Power Plant", *IGCAR, Kalpakkam*.
- [4] T. Jayanthi et al. "Simulation and Integrated Testing of Process Models of PFBR Operator Training Simulator", *ELSEVIER Energy Procedia*, Volume 7, 2011.
- [5] Jaakko Meittinen, "Nuclear Power Plant Simulators: Goals and Evolution", *THICKET 2008, Session III, Paper 07*, May 2008.
- [6] H.Seetha et al., "Visualization of Plant dynamics using soft screens for PFBR Operator Training Simulator", *SSN: 2277-128X (Volume-7, Issue-6)*, June, 2017.
- [7] N Jasmine et al., "Simulation of control logics for plant transition state for PFBR operator training simulator", *International Conference on recent trends in information technology*, 2011.
- [8] "Role of Simulators in Operator Training". *Nuclear Energy Agency, NEA/CSNI/R(97)13*, June, 1998

- [9] W. J. Vaudrey, "Simulator training for nuclear reactor plant operators", IEE Colloquium on operator training simulators, 1992.
- [10] M. Dixon, "Training Simulators", IEE Colloquium on operator training simulators, 1992.
- [11] J. van Loon, "The use of dynamic process models for operator training", IEE Colloquium on operator training simulator, 1992.
- [12] John Jacob Adams, "A Nuclear Power Plant Simulator", *University of Central Florida*, 1973.
- [13] Ann MaQuaid. "Dear Valued AlphaServer Customer", April 28, 2007. *Hewlett-Packard Company*.
- [14] T. Jayanthi et al., "Verification and Validation of simulated process models of KALBR-SIM" training simulator, *International journal of humanities and social sciences*, 2015.
- [15] T. Jayanthi et al., "Simulation and integrated testing of process models of PFBR operator training simulator", *Asian Nuclear Prospects 2010 by Elsevier Publications*.
- [16] Pedro A. Corcuera, "A Full scope nuclear power plant training simulator: Design and Implementation experiences", *University of Cantabria*, 2003.
- [17] L3 MAPPS, "Simulators for nuclear power plant engineering", White Paper, *May 2013*.
- [18] IAEA-TECDOC-1411, "Use of control room simulators for training of nuclear power plant personnel", *September 2004*.
- [19] N. E. Bush, "Training simulators for nuclear power plant reactor operator", *American Institute of Electrical Engineers*, December 1959.

- [20] The CentOS Project, Redhat, "<http://www.centos.org>".
- [21] Haavard Nord, Qt Project, "<http://qt-project.org>", *The Qt Company*.
- [22] D. Richard Hipp, SQLite, "<https://www.sqlite.org>".
- [23] [24] Himanshu Jain, "Verification using satisfiability checking, predicate abstraction and, craig interpolation", Carnegie Mellon University, September 2008.
- [24] Lars Engefield, Callback Functions, "<https://www.newty.de/fpt/callback.html>".
- [25] N. Skoric, A. Kavcic, B. Grobelnik, "Nuclear power plant simulator for general public", Croation nuclear society, Croatia.
- [26] V.V. Korolev, I.I. Sidorova, "Electrical simulation of nuclear reactors", In the soviet journal of atomic energy, Issue 1, pp 837-851, July 1958.
- [27] Jay Jay Billings et al., "A domain-specific analysis system for examining nuclear reactor simulation data for light water and sodium cooled fast reactor", Elsevier Volume 85, pp 856-868, November 2015.
- [28] Ronald Boring, Vivek agarwal, "Digital full-scope simulation of a conventional nuclear power plant control room", Idaho national laboratory, March 2013.
- [29] Daemon Definition, "<http://www.linfo.org/daemon.html>", August 16, 2005
- [30] T. Voggenberger, D. Beraha and F. Cester, "Nuclear power plant simulation and safety analysis", 85748 Garching, Germany, 1993.
- [31] Silberschatz, et al., Operating system concepts, 9th Edition, 2012.
- [32] Andrew S. Tanenbaum, Modern operating systems, 4th Edition, 2014.
- [33] Albert S. Woodhull and Andrew S. Tanenbaum, Operating systems: Design and Implementation, 3rd Edition, 2016.
- [34] Alan Beaulieu, Learning SQL, Oreilly & Associates Incorporated, 2nd edition, 2005.

- [35] Jay A. Kreibich, using SQLite: Small.Fast.Reliable, O'Reilly & Associates Incorporated, 1st Edition, 2010.
- [36] Richard Stallman, "GNU General Public License", Version 3, 29 June, 2007
- [37] GNU Project, Free Software Foundation, "GNU Lesser General Public User License", Version 3, 29 June, 2007
- [38] Linux System Programming, Robert Love, *O'Reilly Media, Chapter 1, 17-35*
- [39] Thomas Cormen, Charles E Leiserson, Ronald Rivest and Clifford Stein
Introduction to Algorithms, 3rd Edition, MIT Press, Chapter 1-3
- [40] Brian Kernighan, Dennis Ritchie, "The C Programming Language"
- [41] Bjarne Stroustrup, "The C++ Programming Language", *Addison-Wesley*
- [42] Ronald Laurids Boring, "Using nuclear power plant training simulators for operator performance and human reliability research", Sandia national laboratories, April 2009.
- [43] Standard review plan, U.S. nuclear regulatory commission, "Reactor Operator training", NUREG-0800, July 1981.
- [44] Using the Meta Object Compiler (moc), "<http://doc.qt.io/archives/qt-4.8/moc.html>"
The Qt Company
- [45] Generic Logic, Inc., GLG Builder and Animation,
"http://www.genlogic.com/doc_html/glgtut.pdf", Version 3.7, November 2017.
- [46] Michael T. Goodrich, Roberto Tamassia, "Data Structures and Algorithms", John Wiley and Sons, 2008
- [47] Avi Silberschatz, Peter Baer Galvin, Greg Gagne, Operating Systems, Inter Process Communication, Yale University, *John Wiley and Sons*, 1992

- [48] Cecil Wayne Ratliff, dBase, "<http://www.dbase.com>".
- [49] John Jacob Adams, "A nuclear power plant simulator", Master Thesis, University of Florida, 1973.
- [50] Nishu, "Development and enhancement of interface framework for the integration of Distributed Digital Control System (DDCS), with PFBR operator training simulator", *Homi Bhabha National Institute, July, 2018*
- [51] Robert F. Rogio, University of North Florida, *Pseudocode Examples*, "www.unf.edu/~broggio/cop2221/2221pseu.htm", *Spring 2015*.